

AD-A152 314

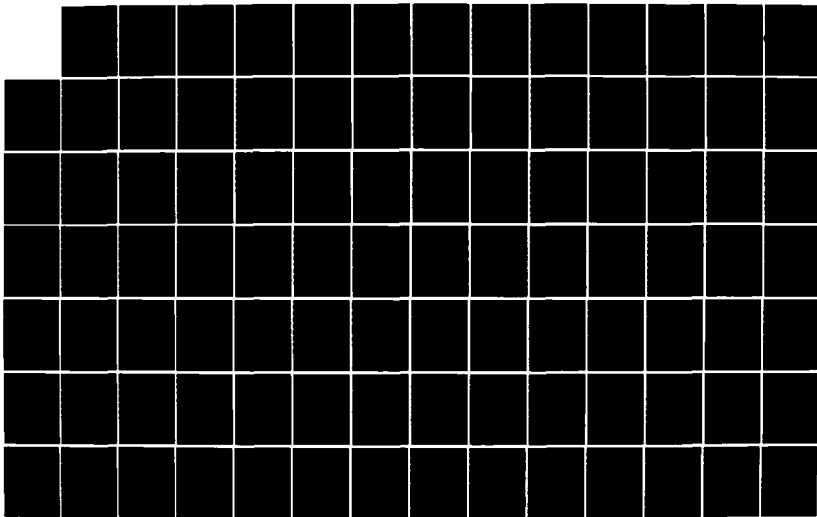
EVALUATION OF ADA (TRADEMARK) AS A COMMUNICATIONS
PROGRAMMING LANGUAGE VOLUME 1(U) SYSCON CORP SAN DIEGO
CA A L BRINTZENHOFF ET AL. 01 MAR 85 DCA100-83-C-0029

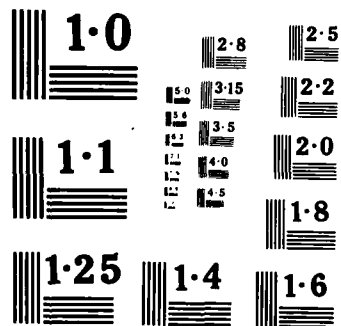
1/3

UNCLASSIFIED

F/G 9/2

NL





Report DCA100-83-C-0029

AD-A152 314

EVALUATION OF ADA* AS A COMMUNICATIONS PROGRAMMING LANGUAGE PHASE II

VOLUME I FINAL PHASE II REPORT

Alton L. Brintzenhoff
Steven W. Christensen
Donald G. Martin
John G. Reddan

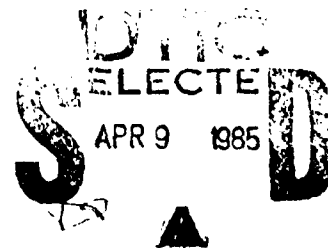
SYSCON CORPORATION
San Diego Division
3990 Sherman Street
San Diego, CA 92110

15 FEBRUARY 1985

Final Report for Period 4 February 1983 - 15 February 1985

APPROVED FOR PUBLIC RELEASE, DISTRIBUTION UNLIMITED

Prepared for
DEFENSE COMMUNICATIONS AGENCY
DEFENSE COMMUNICATIONS ENGINEERING CENTER
1860 Wiehle Avenue
Reston, VA 22090



*Ada is a Registered Trademark of the U.S. Government (Ada Joint Program Office)

85 3 25 096

DTIC File copy

**EVALUATION OF ADA*
AS A
COMMUNICATIONS
PROGRAMMING LANGUAGE
PHASE II**

**VOLUME I
FINAL PHASE II REPORT**

Alton L. Brintzenhoff
Steven W. Christensen
Donald G. Martin
John G. Reddan

SYSCON CORPORATION
San Diego Division
3990 Sherman Street
San Diego, CA 92110

15 FEBRUARY 1985

Final Report for Period 4 February 1983 - 15 February 1985

APPROVED FOR PUBLIC RELEASE, DISTRIBUTION UNLIMITED

Prepared for
DEFENSE COMMUNICATIONS AGENCY
DEFENSE COMMUNICATIONS ENGINEERING CENTER
1860 Wiehle Avenue
Reston, VA 22090

*Ada is a Registered Trademark of the U.S. Government (Ada Joint Program Office)

APR 9 1985
A

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER DCA100-83-C-0029	2. GOVT ACCESSION NO. AD A152314	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) EVALUATION OF ADA AS A COMMUNICATIONS PROGRAMMING LANGUAGE - PHASE II		5. TYPE OF REPORT & PERIOD COVERED Final Phase II Report 4 Feb 83 - 15 Feb 85
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Alton L. Brintzenhoff, Steven W. Christensen, Donald G. Martin, John G. Reddan		8. CONTRACT OR GRANT NUMBER(s) DCA100-83-C-0029
9. PERFORMING ORGANIZATION NAME AND ADDRESS SYSCON Corporation (San Diego Division) 3990 Sherman Street San Diego, CA 92110		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 33126K, 1053(B461), 351C,-
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Communications Engineering Center Code R620 1860 Wiehle Avenue Reston, VA 22090		12. REPORT DATE 1 March 1985
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE --
16. DISTRIBUTION STATEMENT (of this Report) Approved for Public Release, Distribution Unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES DCEC Contract Officers Representatives Mr. Paul M. Cohen, Mr. John Nowakowski		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Ada, Communications Protocols, Transmission Control Protocol (TCP), Internet Protocol (IP), Advanced Data Communications and Control Procedures (ADCCP), Trusted Software, Advanced Command and Control Architectural Testbed (ACCAT) GUARD, software development/performance quality factors, Ada-based design methodology, object-oriented design, Ada program design language (PDL), ANNA		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This report documents the results of the Evaluation of Ada as a Communications Programming Language. The overall objectives of the Defense Communications Agency are to evaluate the ability of Ada to effectively implement communica- tions protocol software and the ability to support the DoD Computer Security Initiative Program with regard to designing and implementing trusted and multi- level secure software. The evaluation context was one of software quality using a set of software quality factors which deal with both software develop- ment and software performance aspects. A large scale software development was		

DD FORM 1473
1 JAN 73EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

emulated through the formation of two mini software development projects using two actual applications with executable code required as one of the end products. The host and target environment consisted of a VAX 11/780 VMS timesharing system and a prototype, partial-implementation Ada compiler. The original set of protocols was the Segment Interface Protocol (SIP) and the Advanced Data Communications Control Procedures (ADCCP) (Mode VI) of the AUTODIN II packet switched network. Subsequently, the SIP and ADCCP (Mode VI) were eliminated and the standardized and published Transmission Control Protocol (TCP) and the Internet Protocol (IP) were incorporated and the ADCCP (Mode VI) was revised to the more standard ADCCP (Asynchronous Balanced Mode (ABM)) protocol. The protocols were implemented in a host-subscriber network architecture with monitoring of resources and injection of errors provided. In the communications protocols area, data transfers across five layers of the OSI architecture were accomplished. This application consisted of 19 virtual packages, 26 library units, 24 secondary units, 8131 Ada statements, 10,309 comments and a total of 23,674 source lines which included the use of approximately 30 tasks. The trusted software consisted of the reimplementaion of a subset of the Advanced Command and Control Architectural Testbed (ACCAT) GUARD application which was a system designed to monitor, sanitize, and "downgrade" the flow of information exchanged between a high (top secret) and low (secret) system via the Upgrade and Downgrade Trusted Processes which were defined in SPECIAL. In general, the full, planned set of capabilities was implemented. This application consisted of 12 virtual packages, 25 library units, 23 secondary units, 6775 Ada statements(;), 9529 comments and a total of 21,305 source lines which included the use of approximately 30 tasks. The software architectures of both applications were evaluated with respect to software development and software performance characteristics. At the beginning of the project, a prototype design methodology was formed and was based on established software engineering principles, and the use of existing generic models such as the ISO Open Systems Interconnections Reference Model and Sublayer models. These elements which included the virtual package concept and both graphical and textual representations of the design with an Ada-based PDL were then organized into the macroscopic/microscopic design methodology. The methodology described above is highly compatible with DOD-STD-SDS. Several general software design guidelines were formed and evaluated. It is shown how the methodology can be adapted for trusted software development by supplementing the Ada PDL of the formal specifications with Annotated Ada (ANNA). The evaluation of the Ada language included the syntax, semantics, implementation dependencies and run-time dependencies. Distinguished as well as problematic features are identified and various recommendations are made. A preliminary set of approximately 30 trusted software implementation restrictions were formed, and approximately 70 programming guidelines were defined and evaluated. Specific suggestions and recommendations are given concerning Ada education, where significant training and experience will be required at several different levels. Both compile-time and run-time error information was acquired, analyzed, and subsequently correlated with the various aspects of the intra- and inter-package architectures to determine the effects of various software architectural choices on errors. Several recommendations are made based on the activities of the project. These recommendations encompass the software development methodology, the use of the Ada language, the software application architectures formed, the implications of compile-time and run-time error analysis, the assignment of values to the necessary software quality factors, and the use of software tools and programming support environments. Finally, recommendations for completing the original architecture as a prerequisite to conducting meaningful performance evaluation are given. Volume II and Volume III contain the software listings of the Communications Protocols and Trusted Software applications, respectively.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

TABLE OF CONTENTS

	<u>PAGE</u>
1 INTRODUCTION	1-1
1.1 PURPOSE	1-2
1.2 SCOPE	1-2
2 SUMMARY	2-1
2.1 PROJECT OVERVIEW	2-1
2.2 SOFTWARE APPLICATIONS OVERVIEW	2-2
2.2.1 Communications Protocols	2-2
2.2.2 Trusted Software	2-3
2.3 SOFTWARE DEVELOPMENT METHODOLOGY	2-4
2.4 ADA LANGUAGE EVALUATION	2-7
2.5 SOFTWARE ARCHITECTURES	2-9
2.6 SOFTWARE PERFORMANCE	2-11
2.7 SOFTWARE ERRORS	2-12
2.8 PROGRAMMING SUPPORT ENVIRONMENT	2-12
2.9 PROJECT RETROSPECTIVES	2-13
2.10 RECOMMENDATIONS	2-15
3 TECHNICAL APPROACH	3-1
3.1 PROJECT OVERVIEW	3-1
3.1.1 Background	3-1
3.1.2 Ada Issues	3-2
3.1.3 General Approach	3-2
3.1.4 Defense Communications Agency Objectives	3-3
3.1.4.1 General Objectives	3-4
3.1.4.2 Specific Communications Protocol Objectives	3-4
3.1.4.3 Specific Trusted Software Objectives	3-4
3.1.4.4 Development Methodology Objectives	3-5

3.2	PROTOTYPE METHODOLOGY FORMULATION	3-5
3.2.1	Requirements Formulation Phase	3-7
3.2.2	Top-Level Design Phase	3-7
3.2.3	Detailed Design Phase	3-8
3.2.4	Code/Debug Phase	3-9
3.3	SOFTWARE DEVELOPMENT AND PROJECT MANAGEMENT	3-9
3.3.1	Software Development Phases	3-9
	3.3.1.1 Ada Indoctrination Phase	3-9
	3.3.1.2 Macroscopic Design Phase	3-9
	3.3.1.3 Microscopic Design Phase	3-11
	3.3.1.4 Code/Debug/Modify Phase	3-11
	3.3.1.5 Integrate/Test Phase	3-11
	3.3.1.6 Development/Performance Evaluation Phase	3-11
3.3.2	Project Management	3-11
	3.3.2.1 Preliminary Design Reviews	3-12
	3.3.2.2 Interim Design Reviews	3-12
	3.3.2.3 Critical Design Reviews	3-12
	3.3.2.4 Progress Reviews	3-13
	3.3.2.5 System Testing Reviews	3-13
3.3.3	Software Development Control	3-13
3.3.4	Integration and Testing Procedures and Standards	3-14
	3.3.4.1 Integration and Testing Overview	3-14
	3.3.4.2 Module Testing Objectives	3-14
	3.3.4.3 System Integration and Testing Objectives	3-15
	3.3.4.4 Test Software Development	3-15

tion For	<input checked="" type="checkbox"/>
ment	<input checked="" type="checkbox"/>
and	<input checked="" type="checkbox"/>
proposed	<input type="checkbox"/>
option	<input type="checkbox"/>
tion/	<input type="checkbox"/>
ability codes	<input type="checkbox"/>
the after	<input type="checkbox"/>



3.4	SOFTWARE QUALITY ASSESSMENT	3-15
3.4.1	Software Quality Factors	3-15
3.4.2	Criteria for Software Quality Factors	3-16
3.4.3	Application-Oriented Requirements	3-21
	3.4.3.1 Communications Application Requirements	3-21
	3.4.3.2 Trusted Software Application Requirements	3-22
3.4.4	Ada Language Characteristics	3-23
3.5	DATA COLLECTION	3-24
3.5.1	Criteria For Data Collection and Evaluation	3-24
3.5.2	Software Architecture Data	3-26
	3.5.2.1 Software System Architecture	3-27
	3.5.2.2 Compilation Unit Architecture	3-27
	3.5.2.3 Compilation Unit Statement Characteristics	3-28
	3.5.2.4 Application-Dependent Architecture Characteristics	3-28
3.5.3	Software Error Data	3-29
	3.5.3.1 Ada Language Errors	3-31
	3.5.3.2 Design Errors	3-32
3.5.4	Programmer Interview Data	3-32
	3.5.4.1 Ada Language	3-33
	3.5.4.2 Methodology	3-33
	3.5.4.2.1 Macroscopic Design Phase	3-34
	3.5.4.2.2 Microscopic Design Phase	3-35
	3.5.4.2.3 Code/Debug Phase	3-36
	3.5.4.2.4 System Integration Phase	3-36
	3.5.4.2.5 Design Guidelines	3-37
	3.5.4.2.6 Software Tools	3-37
	3.5.4.3 Project/Application Evaluation: Alternatives/Retrospectives	3-37
	3.5.4.3.1 Communications Protocols	3-38
	3.5.4.3.2 Trusted Software	3-38

3.5.5	Software Performance Data	3-39
3.5.5.1	Application Architecture	3-39
3.5.5.1.1	Communications Protocols	3-39
3.5.5.1.2	Trusted Software	3-39
3.5.5.1.3	General Performance	
	Considerations	3-41
3.5.5.2	Ada Language Issues	3-41
3.5.5.3	Programming Support Environment	
	Issues	3-42
4	ANALYSIS	4-1
4.1	SOFTWARE DEVELOPMENT METHODOLOGY ANALYSIS	4-1
4.1.1	Overview	4-1
4.1.2	Macroscopic Design Phase	4-2
4.1.2.1	Virtual Package Concept	4-2
4.1.2.2	Object Oriented Design Diagrams	4-3
4.1.2.3	Macroscopic PDL	4-5
4.1.3	Microscopic Design Phase	4-6
4.1.4	Code/Debug Phase	4-7
4.1.5	System Integration Phase	4-8
4.1.6	Design Guidelines	4-8
4.1.7	Programming Guidelines	4-9
4.1.8	General Design Methodology Factors	4-10
4.1.8.1	PDL Characteristics	4-10
4.1.8.2	DOD-STD-SDS Compatibility	4-13
4.1.9	Application Dependent Methodology	
	Characteristics	4-15

4.1.9.1	Communications Protocols	4-15
4.1.9.1.1	Transmission Control Protocol (TCP) and Inter- net Protocol (IP) Specifications Issues	4-15
4.1.9.1.2	Transmission Control Protocol (TCP) and Inter- net Protocol (IP) Transition Issues	4-16
4.1.9.2	Trusted Software	4-19
4.1.10	Software Tools	4-20
4.2	ADA LANGUAGE EVALUATION	4-21
4.2.1	Ada Language Factors	4-21
4.2.2	Ada Education Factors	4-25
4.3	SOFTWARE ARCHITECTURE ANALYSIS	4-26
4.3.1	Software System Architecture	4-26
4.3.1.1	Communications Protocols Software Architecture Analysis	4-26
4.3.1.1.1	Inter-Virtual Package Architecture Analysis	4-29
4.3.1.1.2	Intra-Virtual Architectural Analysis	4-31
4.3.1.2	Trusted-Software Software Architecture Analysis	4-32
4.3.1.2.1	Inter-Virtual Package Architecture Analysis	4-41
4.3.1.2.2	Intra-Virtual Package Architecture Analysis	4-42
4.3.2	Compilation Unit Architecture	4-43
4.3.2.1	Communications Protocols System Compilation Unit Statement Characteristics	4-43
4.3.2.2	Trusted Software Compilation Unit Architecture	4-44

4.3.3	Compilation Unit Statement Characteristics	4-46
4.3.3.1	Communications Protocols System Compilation Unit Statement Characteristics	4-46
4.3.3.2	Trusted Software Compilation Unit Statement Characteristics	4-49
4.4	SOFTWARE PERFORMANCE ANALYSIS	4-49
4.4.1	General Performance Characteristics	4-49
4.4.2	Communications Protocols Performance Characteristics	4-52
4.4.3	Trusted Software Performance Characteristics	4-52
4.5	SOFTWARE ERROR ANALYSIS	4-53
4.5.1	Compilation Errors	4-53
4.5.2	Execution Errors	4-55
4.5.3	Software Error-Architecture Correlation	4-56
	4.5.3.1 Communications Protocol	4-56
	4.5.3.2 Trusted Software	4-57
4.6	PROGRAMMING SUPPORT ENVIRONMENT	4-57
4.6.1	Compile-Time Environment	4-57
4.6.2	Run-Time Environment	4-59
5	CONCLUSIONS/RESULTS	5-1
5.1	SOFTWARE DEVELOPMENT METHODOLOGY	5-1
5.1.1	Macroscopic Design Phase	5-1
5.1.2	Microscopic Design Phase	5-3
5.1.3	Code/Debug	5-4
5.1.4	System Integration	5-4
5.1.5	Design Guidelines	5-6
5.1.6	Programming Guidelines	5-6
5.1.7	General Software Development Methodology Considerations	5-7
5.1.8	Application-Dependent Characteristics	5-9
	5.1.8.1 Communications Protocols	5-9
	5.1.8.2 Trusted Software	5-10

5.2	ADA LANGUAGE EVALUATION	5-10
5.2.1	Ada Language Syntax and Semantics	5-10
5.2.2	Ada Language Education	5-13
5.3	SOFTWARE ARCHITECTURE	5-14
5.3.1	Communications Protocols System	5-14
5.3.1.1	Inter-Virtual Package Analysis Summary	5-14
5.3.1.2	Intra-Virtual Package Analysis Summary	5-17
5.3.1.3	Compilation Unit Analysis Summary	5-19
5.3.1.4	Compilation Unit Statement Characteristics	5-19
5.3.1.5	Other Observations	5-22
5.3.2	Trusted Software System	5-25
5.3.2.1	Inter-Module Architectural Analysis Summary	5-25
5.3.2.2	Intra-Virtual Package Architectural Analysis Summary	5-27
5.3.2.3	Compilation Unit Architecture Analysis Summary	5-29
5.3.2.4	Compilation Unit Statement Characteristics	5-31
5.3.2.5	Other Observations	5-31
5.4	SOFTWARE PERFORMANCE	5-32
5.5	SOFTWARE ERRORS	5-33
5.6	PROGRAMMING SUPPORT ENVIRONMENT	5-34
5.6.1	Compile-Time Environment	5-34
5.6.2	Run-Time Environment	5-35

6	RECOMMENDATIONS	6-1
6.1	SOFTWARE DEVELOPMENT METHODOLOGY	6-1
6.1.1	Macroscopic Design Methodology	6-1
6.1.2	Microscopic Design Methodology	6-2
6.1.3	Code/Debug	6-2
6.1.4	Integrate/Test	6-3
6.1.5	PDL Considerations	6-3
6.1.6	Software Tools Recommendations	6-3
6.1.7	Trusted Software Development Methodology	6-4
6.2	ADA LANGUAGE	6-5
6.2.1	Ada Language Features	6-5
6.2.2	Ada Language Education	6-7
6.3	SOFTWARE ARCHITECTURES	6-7
6.3.1	Communications Protocols	6-8
6.3.2	Trusted Software	6-9
6.4	SOFTWARE PERFORMANCE	6-9
6.5	SOFTWARE ERRORS	6-10
6.6	PROGRAMMING SUPPORT ENVIRONMENT	6-11
6.6.1	Compile-Time Environment	6-11
6.6.2	Run-Time Environment	6-11
6.6.3	Programming Support Environment	6-12
6.7	PROJECT RECOMMENDATIONS	6-12
7	REFERENCES	7-1
7.1	Military Standards and Specifications	7-1
7.2	System Specifications and References	7-2
7.3	Other Government References	7-3
7.4	Nongovernment References	7-3

usable Ada subset given the restrictions imposed. Implementing these restrictions requires both syntactic and semantic analysis and thus some type of Ada preprocessor or modified compiler front end will be required.

In the area of Ada education, significant training and experience will be required at several different levels. Managers will need to be aware of the increased emphasis placed on software design and this emphasis will be even greater to produce transportable and reusable software. To use many of the Ada features effectively, the developer will need to have a strong software engineering orientation. Although Ada has many excellent features, Ada usage will become truly effective and productivity will increase only when software tools become available which directly support a given software methodology and Ada users become familiar with the methodology and the supporting tools. The existing Reference Manual for the Ada Programming Language /M18183/ is an acceptable document for compiler writers. However, it presents significant usability problems in terms of learning Ada, especially in understanding the language complexities and subtleties. A recommendation is that an abridged reference manual be formed which emphasizes the developer's usability of the document. The Rationale for the Ada Programming Language /H0NE84/ should become an integral part of any Ada education effort. It is clear that four to six weeks of dedicated education may be required to produce effective and efficient programmers; a one-week, syntax-oriented Ada class will not be effective in producing programmers who can apply Ada in a software engineering context.

2.5 SOFTWARE ARCHITECTURES

In the communications protocols application, an early objective was to incorporate the principles of the Open Systems Interconnection (OSI) Reference and Sublayer models. They were incorporated into the software designs at the virtual package

calls, representation capabilities, and many of the pragmas germane to communications applications were not available or were partially implemented. With the Ada features used, there were no specific problems other than those summarized below.

Exceptions appear to be problematic from several points of view. Exactly how, when, and where they should be used, and how to include the management of both standard and user-defined error conditions and required processing as an integral part of the design are problem examples. In package specifications, exceptions are "weakly" associated with their source, with the result that exception handlers may be needlessly proliferated.

Somewhat more problematic is dealing with implementation-specific dependencies and run-time support mechanizations which are not specified as part of the Ada standard. Examples of such problems are package size limitations, task stack size limitation (both of which can severely impact architectural considerations), algorithms selected for task context switching and selective wait mechanization, and whether or not generic instantiations share bodies when data representations are the same at the machine level. It is necessary to obtain a validated, evaluated compiler in which the evaluation criteria have been derived from the application for which the compiler is targeted.

Approximately 70 programming guidelines were defined which were evaluated at the conclusion of the effort. The guidelines were effective when they were used; however, judgement in their use is still required in some instances.

In addition to the programming guidelines, approximately 30 trusted software implementation restrictions were formed. These were formed in the context of producing software which would be rated high with regard to the software quality factors of Maintainability, Testability, Correctness, Integrity, Reliability, Robustness, formal verifiability and retaining a

that a set of software quality weights be defined as part of the requirements to assure that the developed software will have the correct software quality characteristics.

Because of the additional requirements for trusted software, especially the formal design and implementation verification associated with the A1 and previous A2 levels of /USD083/, the methodology was adapted to the development of trusted software. The essential difference is that early in the requirements phase of the development cycle, the trusted and nontrusted software were separated and permitted to proceed along parallel paths. The only difference between the two paths is that the trusted path will use the Formal Top-Level Specifications (FTLS) and Descriptive Top-Level Specifications (DTLS). These may be refined into more detailed specifications at each level of design to produce what are generally referred to as FnLS and DnLS. Another key difference is that, in producing the lower level FnLS and DnLS specifications in Ada PDL, the designs are supplemented with Annotated Ada (ANNA) /KRIE83/ and /LUCK84/ to provide more complete and precise descriptions of the trusted software.

During the course of the project, a draft version of DOD-STD-SDS /DSDS83/ was reviewed. It was concluded that the methodology is highly compatible with DOD-STD-SDS, since the macroscopic designs correspond to the Top-Level Design Specification (C5A), and the microscopic designs correspond to the Detailed Design Specification (C5B). Other correspondences exist at the configuration management level.

2.4 ADA LANGUAGE EVALUATION

The Ada language has excellent features for producing modular software and providing multiple levels of design abstraction for the types of applications considered. Unfortunately, due to the compiler used, the full power of Ada was not available. Separate compilation, generics, task types, timed and conditional entry

The methodology has worked very well both for the initial designs and for assisting in making the software changes which resulted from the transition to TCP, IP and ADCCP (ABM) from the original SIP/ADCCP (Mode VI) protocols.

Several issues were raised during the use of this design methodology. The transition from the macro to the micro and the micro to the code encountered some difficulty in that some components had been over-designed while others had been under-designed. Either case is undesirable since detailed design or coding should not be done in the macro design phase and neither should macro design be done in the coding phase. This problem may be reduced by using design reviews, walkthroughs and possibly by obtaining statistics for PDL expansion ratios. Another key aspect is to have the requirements/macro design phases, the macro/micro design phases, and the micro/code-debug phases overlap. As each lower level of detail is explored and refined, it is possible to make corrections or improvements to the preceding higher level of design or requirements prior to a full commitment to next-level details. This is similar to the rapid prototyping concept which permits alternatives to be explored rather than initially committing to a single idea. The concept of rapid prototyping, including the use of a supporting executable PDL, should be provided for in the methodology.

Several design guidelines were formed and evaluated. To make the guidelines more complete, they should be compatible with or refer to the set of programming guidelines which will be used, they should refer to particular application-dependent criteria which may influence the overall architectures, and they should include specific transportability and reusability requirements applicable to the developed software.

Many different types of architectures can be formed in Ada, each meeting the basic set of requirements but with rather different coincidental characteristics. Because of this, it is essential

virtual packages and which will be "hidden" and used to support the visible packages.

The second step consists of organizing the actual compilation or library units within each virtual package and indicating their interdependencies in terms of control flow among the visible entities of each Ada package. This results in the object oriented design diagrams which are similar to those of /BOOC83/ and /BUHR84/. The overall software architecture of the system will have been defined at this point such that considerable design visibility exists without commitment to significant detail.

In the third step, the diagrams are converted into the corresponding macroscopic PDL which allows refinements such as completed data types, specification of formal arguments for generics, subprograms and task entries, the description of major logic decisions and data types within visible entities of packages, the declaration of lower level hidden entities and the use of embedded English language statements to indicate details which are to be converted to Ada source code.

In the fourth step all library units are compiled as a step in verifying the correctness of the designs and achieving an initial step toward system integration.

The microscopic design phase is similar except that more details are added to existing units by either converting existing embedded English statements into Ada source code or into more refined statements. Secondary and tertiary units which were previously only declared are now expanded into their bodies and may indicate still lower levels of nested support units.

The code/debug phase deals with the conversion of the microscopic designs into Ada source code and production of completed virtual packages which are integrated with other virtual packages to form the final system.

2.3 SOFTWARE DEVELOPMENT METHODOLOGY

At the beginning of the project, approximately one month was devoted to Ada indoctrination, which included a review of the Ada language, especially the more advanced features such as data types, tasks, generics, and exceptions as well as a review of the issues associated with the formulation and use of program design languages. Based on the work of Grady Booch in /BOOC83/, and a presentation at a SIGAda meeting by Dr. R.J.A. Buhr, whose design approach is now documented in /BUHR84/, a prototype design methodology was formed. The principles included in the design methodology were: to use established software engineering principles, establish an early Ada orientation with late commitment to Ada details, provide early and continual design visibility, provide for design continuity across the various software development phases, provide a basis for configuration management, and to be able to incorporate existing generic models. Because of the desire to have an intermediate level of design abstraction between Ada packages and the Ada program library, the virtual package concept was formed. These elements were then organized around the use of graphical and textual representations of the design with an Ada-based PDL as the means for refinement. This resulted in the formulation of the macroscopic/microscopic design methodology.

The macroscopic design phase consists of four steps which are summarized below and assume that the application requirements already exist. The first step is to divide the entire set of requirements into a collection of virtual packages which represent major functional entities. The use of the virtual package accomplishes two goals. It precludes having to deal with the many packages (possibly hundreds) which may result in a large system at the beginning of a design. Since a virtual package is similar to an Ada package, it permits a design to show readily, via an architecture other than nested Ada packages, which components will be exported for use by components of other

(Mode VI) to the more standard ADCCP (Asynchronous Balanced Mode (ABM)) protocol. A host-subscriber type network was established in which both suitable system management functions, similar to an actual system, and capabilities to inject various types of protocol-related error conditions would be implemented.

2.2.2 Trusted Software

In the trusted software application, the original Advanced Architectural Command and Control Testbed (ACCAT) GUARD /WOOD78/, /LOGI79A/, /LOGI79B/, /BALD79/ is designed to provide secure, monitored, controlled transfer of data between a high-level (TOP SECRET) and a low-level (SECRET) system. Separation of high-level and low-level entities (files, queues) is maintained by use of the Kernelized Secure Operating System (KSOS). To accomplish the intersystem transfer of data, the high-level and low-level software in the KSOS GUARD system is interfaced by two trusted processes. The Upgrade Trusted Process (UGTP) is responsible for transferring low-level information to the high-level system; the Downgrade Trusted Process (DGTP) is responsible for transferring high-level information to the low-level system under the control of Sanitization Personnel (SP) and a Security Watch Officer (SWO). The KSOS was used for all high-low and low-high message transfers by the trusted processes. Other adjunct routines were defined to deal with UNIX interprocess communication via ports. Since the communications interfaces with the ARPANET were not an area of special concern with respect to trusted software, they were simulated with elementary CRT man-machine interfaces. The SWO and the SP interfaces were preserved as originally specified. Because of the dependency of the trusted software, which was formally specified in SPECIAL /CHEH80/, on the use of KSOS executive service calls, this interface was preserved and the functionality of KSOS was emulated.

Because the Ada language represents a new tool, derived in part from software engineering principles and considerations, it appears that overall software quality is the real issue addressed by the introduction of Ada. Therefore, the emphasis is on conducting this evaluation in the context of software quality. A set of software quality factors, concerning both development and performance aspects, based on work in /COOP79/, formed the basis for the evaluation. These software quality factors are the frequently discussed quantities of Efficiency-I (language expressability), Flexibility, Interoperability, Maintainability, Reusability, Testability and Transportability in the software development area; and Correctness, Efficiency-II (execution efficiency), Integrity, Reliability, Robustness and Usability in the software performance area.

In both applications the requirement was to produce executable code as a means of obtaining firsthand experience in the development aspects of Ada features and with the performance aspects of the implemented software. Because of this, the project exhibited a considerable degree of realism.

2.2 SOFTWARE APPLICATIONS OVERVIEW

2.2.1 Communications Protocols

The original requirements were to implement the Segment Interface Protocol (SIP) and the Advance Data Communications Control Procedures (ADCCP) (Mode VI) of the AUTODIN II /WEST78/, /WEST79/ packet switched network as a way of evaluating the use of Ada in this type of application. With the demise of the AUTODIN II network and the publication of the Transmission Control Protocol (TCP) /M17883/ and the Internet Protocol (IP) /M17783/ standards, the project, which had been active for approximately ten months, was directed to terminate activities on the SIP and ADCCP (Mode VI) and begin the necessary redesign to implement the TCP and IP protocols. A decision was then made to revise the ADCCP

SECTION 2

SUMMARY

2.1 PROJECT OVERVIEW

The objectives of the Defense Communications Agency in the evaluation of Ada as a communications programming language are to evaluate the ability of Ada to effectively implement communications protocol software and to support the DOD Computer Security Initiative Program with regard to designing and implementing trusted and multilevel secure software. In both application areas, the objectives are to evaluate the use of Ada for these types of applications, identify any problems or other factors which need to be considered, and to recommend a suitable Ada development methodology.

The communications protocols application objectives are to develop embedded software that would improve the software quality characteristics of transportability, reusability, maintainability and reliability. Finally, the objectives are to determine how to achieve effective use of Ada, decrease software development time, provide more accurate development results, and provide a more reliable means of achieving the results.

Due to the overriding importance of correctness, integrity and reliability of trusted software, the robustness of the features of the Ada language, and the corresponding capability of producing very complex designs, limitations may have to be placed on the use of Ada in the development of trusted software. The emphasis is to explore the use of various Ada features used for this type of application and determine whether these features result in verifiable designs and code, whether specific programming guidelines which proscribe or prescribe the use of certain features are required, and how such restrictions should be implemented and enforced.

subsections are: 1) Software Development Methodology Evaluation, 2) Ada Language Evaluation, 3) Software Architecture Evaluation, 4) Software Performance Evaluation, 5) Software Error Evaluation, 6) Programming Support Environment Implications, and 7) Project Retrospectives. These primary areas provide the basic information for making recommendations in these respective areas. In addition, application-specific data, information and results are included for the communications protocols and trusted software applications.

SECTION 1 INTRODUCTION

1.1 PURPOSE

This Final Phase II Report for the Evaluation of Ada as a Communications Programming Language documents the findings of a two-year project designed to assess the effectiveness of the use of Ada as a communications programming language. Two types of communications applications were examined: a communications protocols application with a simulated network architecture, and a trusted software application designed to arbitrate the flow of messages between a top secret and a secret system.

Volume I of this Report 1) defines the software applications that were implemented, 2) identifies the technical approach that was taken in collecting and analyzing the data, 3) establishes the criteria for evaluating the results of the project, 4) identifies the analysis that was performed on the data and information that were produced or derived, 5) documents a set of conclusions/results based on the criteria, data and analysis, 6) makes recommendations based on the conclusions and results, and 7) provides a summary of the conclusions, results, recommendations, and project retrospectives.

Volume II, Final Phase II Report: Communications Protocols Application, and Volume III, Final Phase II Report: Trusted Software Application, include design diagrams, Ada source-code listings, and a summary User Manual for the respective applications.

1.2 SCOPE

This report covers all phases of the project and addresses all results, both positive and negative, that were identified throughout the entire project. The major topics of the various

LIST OF TABLES

TABLE		PAGE
3.4-1	Software Development Quality Factors	3-17
3.4-2	Software Performance Quality Factors	3-17
3.4-3	Criteria for Software Quality Factors	3-18
3.4-4	Specific Performance Requirements	3-22
3.4-5	General Performance Requirements	3-22
3.5-1	Macroscopic Ada PDL Criteria	3-35
3.5-2	Software Performance Criteria	3-40
3.5-3	Class A1 - Verified Design Criteria	3-41
4.1-1	Software Design Methodology Objectives	4-1
4.1-2	PDL Expansion Ratios	4-12
4.2-1	Software Application-Dependent Performance Requirements	4-24
4.3-1	Composite Software Development Statistics	4-43
4.3-2	Communications Protocols System Software Statement Analysis Summary	4-45
4.3-3	Trusted Software System Software Statement Analysis Summary	4-47
4.3-4	Communications Protocols System Aggregate Statement Statistics	4-48
4.3-5	Trusted Software Aggregate Statement Statistics	4-50
4.5-1	Compilation-Related Errors	4-54
5.3-1	Software Development Statistics	5-21
6.1-1	Software Tool Recommendations	6-4

LIST OF ILLUSTRATIONS

FIGURE		PAGE
3.2-1	Software-Development-Phase Notations	3-6
3.3-1	Software Development Phases	3-10
3.4-1	Software Quality Factor-Criteria Interrelationships	3-20
3.5-1	Ada, Methodology and Architecture Evaluation Components	3-25
3.5-2	Software Structure/Error Analysis	3-30
4.1-1	Communications Protocols Detailed Architecture Transition Components	4-18
4.3-1	System Architecture Model Development	4-27
4.3-2	Detailed Architecture Development	4-28
4.3-3	Communications Protocols System Detailed Architecture	4-30
4.3-4	Host_TCP_Server Virtual Package Diagram	4-33
4.3-5	Original ACCAT GUARD System Configuration	4-34
4.3-6	ACCAT GUARD Software	4-35
4.3-7	Modified GUARD Configuration	4-36
4.3-8	Trusted Software System Detailed Architecture	4-37
4.3-9	GUARD Message Flow	4-38
4.3-10	GUARD Transaction Flow	4-39
4.3-11	Downgrade Trusted Process Interactions	4-40
5.1-1	Methodology Compatability with DOD-STD-SDS	5-2
5.1-2	Trusted Software Design Methodology	5-11
5.3-1	Communications Protocols System Inter-Virtual Package Analysis Summary	5-15
5.3-2	Communications Protocols System Intra-Virtual Package Analysis Summary	5-18
5.3-3	Communications Protocols System Compilation Unit Analysis Summary	5-20
5.3-4	Trusted Software Inter-Virtual Package Analysis Summary	5-26
5.3-5	Trusted Software Intra-Virtual Package Analysis Summary	5-28
5.3-6	Trusted Software Compilation Unit Analysis Summary	5-30

APPENDICES

		<u>PAGE</u>
A	SOFTWARE DEVELOPMENT GUIDELINES	A-1
B	ADA RESTRICTIONS FOR TRUSTED SOFTWARE IMPLEMENTATION	B-1
C	SOFTWARE TOOL RECOMMENDATIONS, DESCRIPTIONS	C-1
D	COMPILER LIMITATIONS AND IMPACTS	D-1

level to capture the transportability and reusability characteristics. Collections of virtual packages were distributed across the application, TCP, IP, ADCCP, pseudolink layers, and the system management services. The organization of the software along these architectural lines significantly facilitated the modifications which were made to the software when the transition from the SIP/ADCCP to the TCP/IP/ADCCP protocols was made. Within a given protocol layer, the software was partitioned into Ada packages which principally followed the OSI sublayer model boundaries and consisted of the service, protocol, access, and intralayer management components. At least one issue which warrants further study with respect to transportability and reusability is the placement of system management functions with two extremes being either totally within the layer or totally outside the layer.

In the trusted software application, the existing UNIX-based architecture was translated into an Ada-based architecture. Several significant changes occurred in making this transition. The entire architecture was translated from a multiprocessing to a multitasking environment; the original processes were reimplemented as Ada tasks; interprocess communication entities, implemented as UNIX ports, were reimplemented as transporter tasks in Ada; the Kernelized Secure Operating System interfaces were preserved and treated as Ada service entities because the Upgrade Trusted Process and the Downgrade Trusted Process, specified in SPECIAL, were directly dependent on KSOS services. A final deviation was that interfaces with the high and low side of the GUARD to the respective high and low systems via ARPANET and crypto devices were emulated as online terminal users to permit messages to be transferred between the high and low sides via the GUARD. No difficulties existed with the reimplementations. However, significant changes had to be made to the software architecture at the intrapackage level to circumvent compiler problems, specifically those associated with task stack size limitations. As a result, extensive stress testing and

meaningful architecture evaluation with regard to AI trusted software evaluation criteria was not possible. Consequently, there may be problems associated with the planned architecture which impact on the Correctness, Integrity, Reliability, and Robustness of the trusted software, particularly in the areas of data flow and covert channels.

2.6 SOFTWARE PERFORMANCE

The objectives in software performance were to assess the Correctness, Efficiency-II, Integrity, Reliability and Robustness of the two applications.

In the communications protocols application, overall assessment of the software performance factors was severely impeded because of compiler problems, specifically problems resulting from the lack of a time-sliced environment, task stack size limitations, somewhat weak use of exceptions and adverse interactions between tasking and TEXT_IO resulting in spurious errors. Several features of the Ada language which could contribute positively to overall performance were either used or would have been used had they been available. These include access variables, unchecked conversion, and use of pragma INLINE if it had been available.

In the trusted software application, again little was accomplished in the performance area because of the revisions to the planned software architecture to achieve an executing program and because of the lack of project time to inject various error conditions and conduct stress testing. Many questions related to overall performance, especially Correctness, Integrity, Reliability, and Robustness, will need to have the original designs restored and implemented and extensive stress testing and error injection performed to fully assess these performance factors.

2.7 SOFTWARE ERRORS

During the early portion of the project, software compilation error information was collected from both application areas. The programming errors were indicative of a lack of familiarity with the syntax of Ada type and object declarations, inattention to the full implications of using a strongly typed language, the failure to include context clauses resulting in numerous undeclared entities, and conflicts in the use of attributes and types. At the module architecture level, a small number of errors resulted in erroneous programs being produced; objects were operated on concurrently by both a task and a procedure from the same package or by two different tasks concurrently without pragma SHARED being declared for the objects in question. This problem represented a more fundamental misunderstanding of the difference in semantics between tasks and subprograms as processing entities. At the system architecture level, especially in the communications protocols application, insufficient attention was given to the use of exceptions and their semantics as an integral component of the overall design; in the trusted software application, possibly because of the use of exceptions in the SPECIAL specifications, exceptions were included much more effectively. Other minor errors occurred during elaboration (improper ordering, access before elaboration) and during execution (uninitialized variables).

2.8 PROGRAMMING SUPPORT ENVIRONMENT

The host programming support environment as well as the target environment consisted of a VAX 11/780 VMS* timesharing system supporting a variety of users. Because of the resources required by the Ada compiler, the online compilations were limited to

*DEC, VAX and VMS are trademarks of Digital Equipment Corp.

small jobs and larger jobs were required to be run in the batch mode. Programmer productivity could have been increased had there been a less fully loaded system available and had there been more software tools available.

Tools which could have been helpful are a PDL processor, source-level debugger, pretty printer, generalized call-graph generator and, most importantly, a validated, full-capability Ada compiler. The software tools which were used included a screen-oriented text editor; SKETCHER, an interactive ASCII graphics editor for producing object-oriented design diagrams; and a prototype, partial-implementation Ada compiler. As more software tools are developed and as larger Ada systems are designed, implemented and debugged, the demands on programming support environment resources will increase substantially.

Both the compile-time and run-time environments permit significant variations in their implementations with respect to the MIL-STD-1815A. It will be necessary to have not only a validated compiler, but also an evaluated one with the evaluation criteria based, in part, on the application to be implemented and the design methodology to be used.

2.9 PROJECT RETROSPECTIVES

In terms of the overall project, there were several major accomplishments and some major and minor disappointments.

In the communications protocols area, data transfers across five layers of the architecture were accomplished, including the opening and closing of connections, and single terminal echo-testing as well as two-terminal interactive testing. Several major capabilities, although not all, within each of the protocols layers were implemented. Particular disappointments were that more of the protocol error processing features could not be implemented in order to test the overall Efficiency-II,

Correctness, Reliability, Integrity and Robustness of the software. This application consisted of 19 virtual packages, 26 library units, 24 secondary units, 8131 Ada statements, 10309 comments and a total of 23674 source lines including approximately 30 tasks. In attempting to use the TCP and IP specifications, considerable insight was gained into how they might be placed online and revised to make the contained PDL, which is strongly Ada-like, more complete, consistent and usable.

In the trusted software application, the full, planned set of capabilities was implemented within limitations imposed by existing compiler problems. The significant accomplishments include the transition from a UNIX-type architecture, the ability to use Ada tasks to achieve a four-terminal interactive system and the capability to accomplish the transfer of messages and transactions between all four operator stations. Disappointments included the inability to fully implement the original set of Ada designs, the inability to evaluate the designs and architectures against the AI trusted software criteria because of the compromises made in the architecture, and the inability to conduct extensive stress testing and code analysis with regard to programming style and formal verifiability. This application consisted of 12 virtual packages, 25 library units, 23 secondary units, 6775 Ada statements, 9529 comments and a total of 21305 source lines including approximately 30 tasks.

Finally, three important policy issues which had an overall influence on the project are summarized below: First, because of the newness of Ada and the desire to fully explore the use of these features in a "real" application, liberal use of Ada features was attempted at all architectural levels. Second, the development emphasis was on achieving execution of the applications, even at the expense of reduced functionality and altered designs rather than on achieving execution of narrowly limited portions of the system which had been fully implemented.

Third, because of the prototype nature of the project and richness of the Ada language, it was difficult at times to maintain the proper balance between design, exploring alternative designs, and selecting one and implementing it.

2.10 RECOMMENDATIONS

Several recommendations can be made based on the activities of the project.

The development methodology should be formalized, adapted to a set of corresponding documentation standards such as DOD-STD-SDS, and augmented with a set of compatible software tools to make the methodology both effective and efficient. Specific design and programming guidelines addressing transportability and reusability of communications protocol applications with respect to overall software architecture considerations should also be formed.

For the Ada language, a set of compiler system evaluation criteria which are driven by application requirements and software development methodology characteristics should be formed and used to evaluate any validated compiler before selection for a given development project. These criteria must address the compile-time and compiler pragmatics parameters as well as the run-time support environment characteristics.

For trusted software development, the designs should be implemented in an Ada/ANNA combination from the beginning to obviate the need for making subsequent translations from another language and dealing with the various translation and interpretation issues. Since ANNA can supply additional semantic information in package specifications, the use of ANNA should also be considered for enhancing protocol specifications either at the specification level, such as the TCP and IP documents, and definitely at the software implementation level.

For Ada education, a solid software engineering basis is required to use many of the Ada features effectively. This foundation must be supplemented with education on the Ada language itself, the use of the planned development methodology, and the use of the supporting software tools.

Software quality weights need to be established as part of the requirements definition effort to assure that the designed software architectures, from the highest level, reflect these requirements. In both applications, however, for different reasons the software architectures as planned should be fully implemented and carefully evaluated to explore alternative designs and what their impacts would be on the development and performance software quality factors.

In software performance, real progress, insights and definitive answers can be obtained only by completing the original applications and conducting fairly extensive stress testing of the systems and evaluating various software alternatives at both the inter- and intra-package levels.

Programming errors can be reduced with a combination of education and experience. Other errors of a more subtle nature such as those occurring at the inter-package architecture level require careful attention to the overall design and the semantics of specific features used such as the combination of global data or exceptions with tasks. These situations may also be aggravated by run-time support environment idiosyncrasies. To the extent that many of Ada's more advanced features such as task types, allocators, generics with parameterized subprograms and nested generics are combined to produce the overall software architecture, it is difficult to speculate on the nature of the development and performance characteristics of the software until additional experience has been gained.

SECTION 3

TECHNICAL APPROACH

3.1 PROJECT OVERVIEW

The project components are presented as originally planned to provide a context for identifying and evaluating the activities that occurred throughout the project and what their impacts were on the final accomplishments. The variations, deviations, and events are presented and evaluated in Section 4, Analysis; Section 5, Conclusions/Results; and Section 6, Recommendations.

3.1.1 Background

SYSCON performed Phase I of this effort; it consisted of evaluating the Ada concurrent programming (tasking) capabilities as related to communications applications, comparing Ada to the CCITT High-Level Language (CHILL) which is used in telecommunications system applications, and formulating a test and evaluation plan as the basis for this Phase II effort.

The availability of the new programming language, Ada, presents opportunities for developing quality software through the use of language features used previously only in research environments. New controls in the form of programming standards and guidelines and new software design and development methodologies are required to maximize the potential for producing quality software. To evaluate the Ada language, and formulate these standards, guidelines and methodologies, the Defense Communications Agency requested that a test and evaluation plan be formed using Ada to implement two prototype communications applications.

The SYSCON-developed evaluation plan established the approaches to be used in designing, developing, and testing the software,

evaluating the efficiency and effectiveness of Ada as used in these applications, and identifying standards, guidelines, and methodologies to assure overall software quality in the use of Ada.

3.1.2 Ada Issues

The existing version of Ada, ANSI/MIL-STD-1815A, has resulted from extensive open review, test, and evaluation by individuals from government, industry, and educational institutions. Despite this review process and the constructive changes which were made to Ada by the time it became MIL-STD-1815A, there are still innumerable issues which can be explored further through actual use of Ada in attempting to solve some real problems.

Based on actual use of Ada for implementations of stand-alone applications, preliminary results indicate that software development approaches which are different from those presently used may be required to effect the optimal use of Ada. These include the use of Ada as a Program Design Language (PDL), changes in the approach to modularization, additional emphasis on data abstraction, and Ada-tasking constructs for developing concurrent processing applications. Another issue is the effect of individual programming styles on the production of quality software, particularly with regard to transportability and maintainability. Although Ada is a rich, powerful, and versatile language which provides the programmer with many opportunities, a final area of concern is how suitable, effective, and efficient the features of Ada are with regard to specific categories of applications and how Ada can effectively provide the basis for a PDL in these categories.

3.1.3 General Approach

As a means of evaluating Ada and addressing other objectives, the Defense Communications Agency, through the Defense Communications

Engineering Center, has selected representatives of two categories of software to be implemented on a prototype basis using Ada. The first category is a communications application, which began with two communications protocols, the Segment Interface Protocol and the Advanced Data Communications Control Procedure (SIP/ADCCP) of the former AUTODIN II packet switched network. Subsequently, the SIP was replaced with a combination of the newly released (August 1983) Transmission Control Protocol (TCP) /M17783B/ and the Internet Protocol (IP) /M17783A/. The ADCCP (Mode VI), which was peculiar to AUTODIN II, was replaced with the more general ADCCP (Asynchronous Balanced Mode (ABM)). The second category is the Advanced Command and Control Architectural Testbed (ACCAT) GUARD trusted software application, which uses the Kernelized Secure Operating System (KSOS) and functions as a trusted process to permit the controlled exchange of information between separate SECRET and TOP SECRET systems. Thus, the software prototyping will serve as the vehicle for using Ada and acquiring the data and information which are needed to address the Defense Communications Agency's objectives.

3.1.4 Defense Communications Agency Objectives

The objectives of the Evaluation of Ada as a Communications Programming Language are divided into four categories. These are objectives which address Ada issues and solutions generically, objectives peculiar to those types of applications that employ standard protocols, objectives unique to trusted software types of applications, and objectives to recommend a software development methodology for developing Ada software in the communications protocols and trusted software application areas.

This project is designed to acquire realistic experience and knowledge that can be used to form an effective Ada-based software engineering methodology and to determine what problem areas will impact communications applications implemented in Ada. The emphasis is on exploring alternative methods and approaches

and identifying both successes and failures to provide a perspective for forming the necessary methodology and software tools.

3.1.4.1 General Objectives

An objective of the software prototyping effort is to investigate typical communications applications and evaluate the use of Ada in such applications. A related objective is to identify any generally interesting results which may apply outside the communications area. Missing or inefficient Ada features that are germane to communications applications will be identified. The use of assembler code and target machine dependencies will be identified. Software architectures and Ada characteristics that have a major influence on developing transportable and reusable software will be identified. The ability of Ada tasking features to be effective, efficient, and to adequately represent real-time, multi-tasking requirements will be evaluated.

3.1.4.2 Specific Communications Protocol Objectives

The TCP, IP and ADCCP protocols specify interfaces across which information must flow. Servicing of information requires concurrent processing for an effective implementation. The Ada-tasking features and mechanism will be evaluated for efficiency and effectiveness. Since it is desirable to transport protocols to other application areas, another objective will be to determine the influence of Ada, Ada software design, and Ada implementations on the transportability of the implementations.

3.1.4.3 Specific Trusted Software Objectives

Trusted software characteristics may require administrative restrictions on the use of Ada features. Such restrictions will be identified along with a strategy for implementing them. Correspondence tests between the Ada source code and the trusted

software requirements will be performed to determine the influence of Ada features and styles on trusted software development. Due to the real-time requirements of this application, objectives similar to those of the TCP/IP/ADCCP application will apply.

3.1.4.4 Development Methodology Objectives

A methodology for developing communications software with Ada as the implementation language will be recommended based upon the results of the prototyping effort. A specific objective within the methodology formulation is the use of Ada or Ada-like components as a Program Design Language (PDL). Acquired information will provide the basis for forming DCA management decisions relating to software standards, conventions, policies, tools, procedures, and directives in the use of Ada.

3.2 PROTOTYPE METHODOLOGY FORMULATION

The methodology will incorporate a combination of two design methods used to produce software designs: the Structured Analysis and Design Methodology (SADM) as summarized in /PRIV82/ and the object-oriented design approaches presented in /BOOC83/ and /BUHR84/. The methodology consists of two phases of design, a code/debug phase, and culminates in the integration/testing of the resulting software. To effectively understand the issues and problems involved in producing software in Ada, four levels of abstraction following the concept formulation will be utilized. Figure 3.2-1 illustrates these levels and the appropriate points where Ada features are involved. These methodologies are applied in a series of stepwise refinements and are augmented with variations on the graphical design notation presented in /BUHR84/.

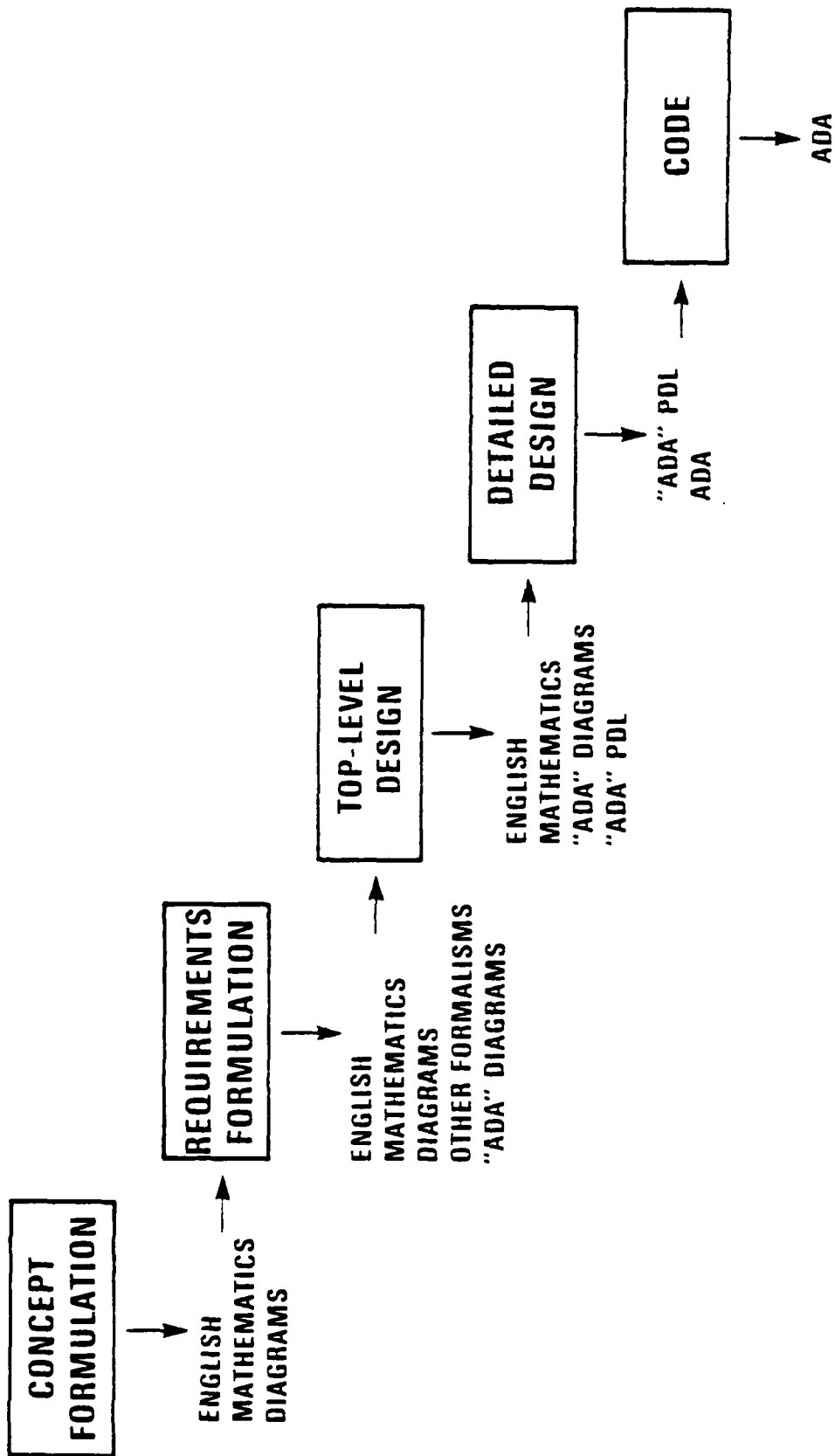


Figure 3.2-1. Software-Development-Phase Notations

3.2.1 Requirements Formulation Phase

This refinement consists of defining system-system interfaces, functionally identifying subsystems and defining inter-subsystem interfaces, intra-subsystem interfaces, and data flow paths. For each subsystem a set of characteristics/requirements will form the basis for refinement into the functional components of each subsystem. This phase culminates with the formation of a detailed architectural design which represents the hierarchical, structural orientations and data dependency relationships of the functional modules.

3.2.2 Top-Level Design Phase

The top-level design phase introduces conceptual components called virtual packages. (In the notation of /M15272/, /M49068/ and /M48379/, these elements correspond to Computer Program Components (CPCs) of Computer Program Configuration Items (CPCIs).) These components are referred to as virtual packages because they exist at a level which is one step higher than actual Ada packages, and they exhibit the characteristics of Ada packages such as having visible and private (internal) components.

The design information will consist of overview diagrams that address interfaces, followed by a Detailed Functional Requirements section for each virtual package. At this level, the SADM approach will be used to form the basic Ada compilation units that make up a virtual package. The visible details of the compilation units will then be documented using object-oriented design diagrams.

The top-level design phase is referred to as the macroscopic design. This step will continue use of specifications to define the operations needed on abstract data types. These operations establish all program units (packages, tasks, subprograms,

3.4.3 Application-Oriented Requirements

There are application-oriented requirements which a language must satisfy to facilitate the development of software for the target applications. This section identifies specific requirements for communications and trusted software applications.

3.4.3.1 Communications Application Requirements

A previous study performed for the Defense Communications Agency /BBNI76/ resulted in the definition of the syntax and semantics of the Communications Oriented Language (COL). As part of that study, three alternative sets of requirements, which are desirable for a COL to have, were examined. The first set was obtained from the "U.S. Air Force HOL Standardization Study"; the second set was obtained from "The Initial Report on the Suitability of JOVIAL for Communications Systems Implementation"; the third set was obtained from "The Rome Air Development Center Report on Common-Communications Processors."

There is commonality among the items of each set; however, there is also some discrepancy. Each list is also a mixture of high-level language-inherent features as well as requirements for access to data, instructions, and controls at the machine level. From these lists a composite of specific requirements, shown in Table 3.4-4, was formed. This list serves as a basis for assessing the efficiency and effectiveness of Ada as a language for developing communications software. The report also indicated some general performance requirements which are shown in Table 3.4-5.

Table 3.4-3. Criteria for Software Quality Factors
(page 2 of 2)

HARDWARE ARCHITECTURE COMPATIBILITY	- The degree to which hardware elements and their configuration are effectively used by an application.
HARDWARE INDEPENDENCE	- The attribute of software that indicates the degree of coupling between the language constructs and the hardware on which the software will operate.
INSTRUMENTATION	- The attribute of software that provides for the control or display of intermediate conditions, events, or data on a conditional or non-conditional basis.
LANGUAGE CONSTRUCTS	- The syntax and associated semantics of the programming language used in the software development.
LANGUAGE IMPLEMENTATION	- The mechanization of the language constructs in a machine representation which can be executed.
MODULARITY	- The attribute of software that provides for the organization of the software into independent cooperating elements.
OPERABILITY	- The attribute of software that determines the type and quantity of user procedures required to operate or interface with the software.
OPERATING SYSTEM ARCHITECTURE COMPATIBILITY	- The degree to which operating system elements, their configuration, and their accessibility are effectively used by applications programs.
OPERATING SYSTEM INDEPENDENCE	- The attribute of software that provides for the minimum direct interaction of developed software with specific operating features.
SELF-DESCRIPTIVENESS	- The attribute of software that provides for clarity and apparentness in describing the purpose or function of the software as well as the algorithm being used and its organization.
SIMPLICITY	- The attribute of software that provides for the implementation in terms most easily understood.
TRACEABILITY	- The attribute of software that provides for logical and structure connectivity from the highest level of specification to the source code implementation.

Table 3.4-3. Criteria for Software Quality Factors
(Page 1 of 2)

ACCURACY - The attribute of software that provides for the usability of the computational results with regard to correctness, precision, and timeliness.

COMMUNICATIONS COMMONALITY - The attribute of software that provides for the use of standard protocols and mechanisms for the interfacing of two software components.

COMMUNICATIVENESS - The attribute of software that provides outputs which can be readily assimilated by a user and requires inputs which can be readily supplied by the user.

COMPLETENESS - The attribute of software that provides for the full implementation of all functions and capabilities specified.

CONCISENESS - The attribute of software that provides for implementation of a function with the use of a minimum quantity of source code.

CONSISTENCY - The attribute of software that provides uniform design and implementation techniques, guidelines, standards, and notation.

DATA COMMONALITY - The attribute of software that provides for the use of standardized data formats and representations.

ERROR MANAGEMENT - The attribute of software to correctly detect, isolate, manage, and inform all specified error conditions.

GENERALITY - The attribute of software that permits it to handle broader scope problems or conditions than those specified.

Table 3.4-1. Software Development Quality Factors

EFFICIENCY I	- A measure of the extent to which algorithms are or can be represented in format using the available language constructs.
FLEXIBILITY	- A measure of the extent to which an operational program can be modified to include new functional capabilities.
INTEROPERABILITY	- A measure of the extent to which two operational programs of different systems can be coupled or interfaced without modification to enhance performance or functional capabilities.
MAINTAINABILITY	- A measure of the extent to which an error in an operational program can be identified, isolated and corrected.
REUSABILITY	- A measure of the extent to which an operational program can be used as a component in another application without modification.
TESTABILITY	- A measure of the extent to which a program can be readily tested to assure that performance criteria are met during the development, maintenance, and modification phases.
TRANSPORTABILITY	- A measure of the extent to which an operational program can be readily transferred to a different hardware or software environment and perform correctly without modification.

Table 3.4-2. Software Performance Quality Factors

CORRECTNESS	- A measure of the extent to which an operational program complies with its specifications, performs its functions and produces acceptable results.
EFFICIENCY II	- A measure of the extent to which an operational program makes optimal use of the system resources including CPU time, memory, and peripherals.
INTEGRITY	- A measure of the extent to which an operation program performs only its intended functions and does not overtly or covertly perform any other functions.
RELIABILITY	- A measure of the extent, with regard to frequency and criticality of failures, to which a program can be expected to perform its required functions in its intended environment.
ROBUSTNESS	- A measure of the extent to which an operational program is able to acceptably manage or respond to conditions outside its intended operational environments.
USABILITY	- A measure of the extent to which program users can prepare input data for, interpret output data from, and control operation of the program and learn to use the program in its intended environment.

and at the lowest level are the measurable parameters which can be related to the software quality criteria.

Software quality is a relative and imprecise entity in that "the degree of excellence" required of software is not absolute. Different organizations and projects may have different objectives. For example, "throw-away" code need be given very little consideration with respect to life-cycle maintainability. Software quality factors such as transportability and efficiency are potentially in conflict and thus necessitate a trade-off or compromise to be achieved.

The primary emphasis of this development effort is to gauge the effect of Ada on both the development cycle of communications software and on the performance aspects of the resulting code. The need is to key on software quality factors which relate to development and performance. Table 3.4-1 lists those software quality factors which relate to or impact on the software development, maintenance, and modification process. Table 3.4-2 lists those software quality factors which relate to or impact on the performance of software implemented in Ada.

3.4.2 Criteria for Software Quality Factors

The criteria identified in Table 3.4-3 represent a set of independent attributes which software may possess both with regard to software development and software performance. An individual criterion may be correlated with more than one software quality factor. The interrelationships between the software quality factors and the software quality criteria are illustrated in Figure 3.4-1. These criteria are taken from /COOP79/ and minor additions have been made.

3.3.4.3 System Integration and Testing Objectives

The objective of the system integration and testing is to combine all software for each application, including the test support software, and exercise the software through the use of the functionally oriented system integration tests. Detailed system integration and testing objectives are identified in MIL-STD-16/9 (NAVY) /M16778/, Section 5.8.3.

3.3.4.4 Test Software Development

Specific requirements for test support software which need to be developed for the TCP/IP/ADCCP and ACCAT GUARD applications will be identified as required. This software will be designed using the macroscopic/microscopic approach established for the application software and will be coded during the code/debug/modify portion of the software development.

3.4 SOFTWARE QUALITY ASSESSMENT

This section identifies the software quality factors and constituent software quality criteria used to evaluate the architectures with respect to the use of the Ada language. Application-specific requirements, which will be used for evaluation, are also listed.

3.4.1 Software Quality Factors

Software quality can be defined as a hierarchical set of software quality parameters. At the highest level is the concept of software quality which is "the composite of all attributes which describe the degree of excellence of computer software" /COOP79/. Next are the conceptual software quality factors which represent the attributes that it is desirable for software to have. At the next lower level are the constituent software quality criteria

Three additional classes of libraries will be used as repositories for storing the System Design, Macroscopic Design, and Microscopic Design documentation.

As a means of documenting program execution errors, a minimal software trouble report system will be used on the VAX. This capability will be implemented using existing software to document corrections and provide a historical record for future analysis.

3.3.4 Integration and Testing Procedures and Standards

This section describes the definition and development of test software and the testing objectives.

3.3.4.1 Integration and Testing Overview

Two levels of testing will be performed during software development. These comprise module testing and system integration testing. The module and system integration testing is performed by the programmer responsible for each application. Test specifications will be provided for each level of testing along with the respective designs.

3.3.4.2 Module Testing Objectives

The objective of the module testing is to exercise each module to assure that all internal program errors have been detected and corrected prior to system integration testing. Detailed module testing objectives are identified in the MIL-STD-1679 (NAVY) /M16778/, Section 5.8.1.

used, and addresses any other outstanding design or Ada programming support environments elements which may influence the implementation. Information gained from the code/debug/modify phase which may have an influence on the designs is evaluated.

3.3.2.4 Progress Reviews

Informal progress reviews with the entire project team will be conducted nominally every two weeks. These reviews are used to exchange ideas, avoid redundant related efforts, and to generally monitor progress and identify problems.

3.3.2.5 System Testing Reviews

During the Integration/Test Phase, a System Testing Review (STR) will be conducted to monitor test/integration results. The purpose is to identify any systematic errors in the designs, implementations or Ada compiler so that corrective action can be taken as early as possible.

3.3.3 Software Development Control

As a means of providing control over and visibility into the software development process and providing control over the status and availability of the software, three classes of program libraries will be used. These are the Development Library, Integration Library, and Release Library. The Development Library will contain all software which is undergoing coding, debugging, or module testing. Upon completion of module testing, the module is moved to the Integration Library for integration. When a cycle of integration has been completed and a portion of the system is ready for use, that portion will be placed in the Release Library. The individual programmers will be responsible for making the transitions under the review of the project manager.

development, and to assure that the application requirements have been met. Reviews provide a means of establishing milestones and evaluating overall progress. Design walkthroughs will be used to facilitate discussion of critical or unclear design elements and of Ada language and Ada PDL issues.

3.3.2.1 Preliminary Design Reviews

Preliminary Design Reviews (PDR) are conducted following the macroscopic designs and approximately two weeks after the initiation of the microscopic design phase. The PDRs will be designed to accomplish several objectives. They will represent a milestone against which progress can be formally measured. They will determine if application requirements have been met, if the macroscopic designs are complete and consistent, and whether the extracted designs are complete and consistent. During the PDRs outstanding problems and design or implementation issues will be identified and reviewed. The PDRs will provide an opportunity to discuss insights gained from the microscopic design phase.

3.3.2.2 Interim Design Reviews

Throughout the project several Interim Design Reviews (IDR) will be held. These serve as formal milestone reviews and as a way of obtaining lessons-learned information which may have a bearing on the remainder of the project. These reviews provide a means for formalized interchanges among management, development, and evaluation personnel.

3.3.2.3 Critical Design Reviews

Following completion of the microscopic designs a Critical Design Review (CDR) will be held. This review evaluates the completeness, correctness and suitability of the microscopic designs, provides a means to identify and resolve any open design issues, evaluates the module and system test specifications to be

3.3.1.3 Microscopic Design Phase

The microscopic design phase consists of refining the macroscopic designs to the microscopic design level. This includes supplying lower level units, more detailed PDL, and converting the PDL of the top-level units into more nearly completed Ada code.

3.3.1.4 Code/Debug/Modify Phase

The code/debug/modify phase consists of converting the microscopic designs into Ada code, debugging the code and forming integrated packages within each virtual package. The modify portion is designed to have each programmer implement a small portion of the other's design as a way of assessing software maintainability issues.

3.3.1.5 Integrate/Test Phase

The integrate/test phase consists of integrating the software associated with the various virtual packages and conducting performance testing on the resultant software.

3.3.1.6 Development/Performance Evaluation Phase

The development/performance evaluation phase consists of defining criteria for evaluation and evaluating the development methodology, the software architectures, the Ada language, the performance of the developed systems and assessing the types of errors encountered and their correlation with the architectures developed during the design process.

3.3.2 Project Management

A part of the development approach is to conduct reviews throughout the project at critical points. The objectives are to discern the difficulties encountered during the design and

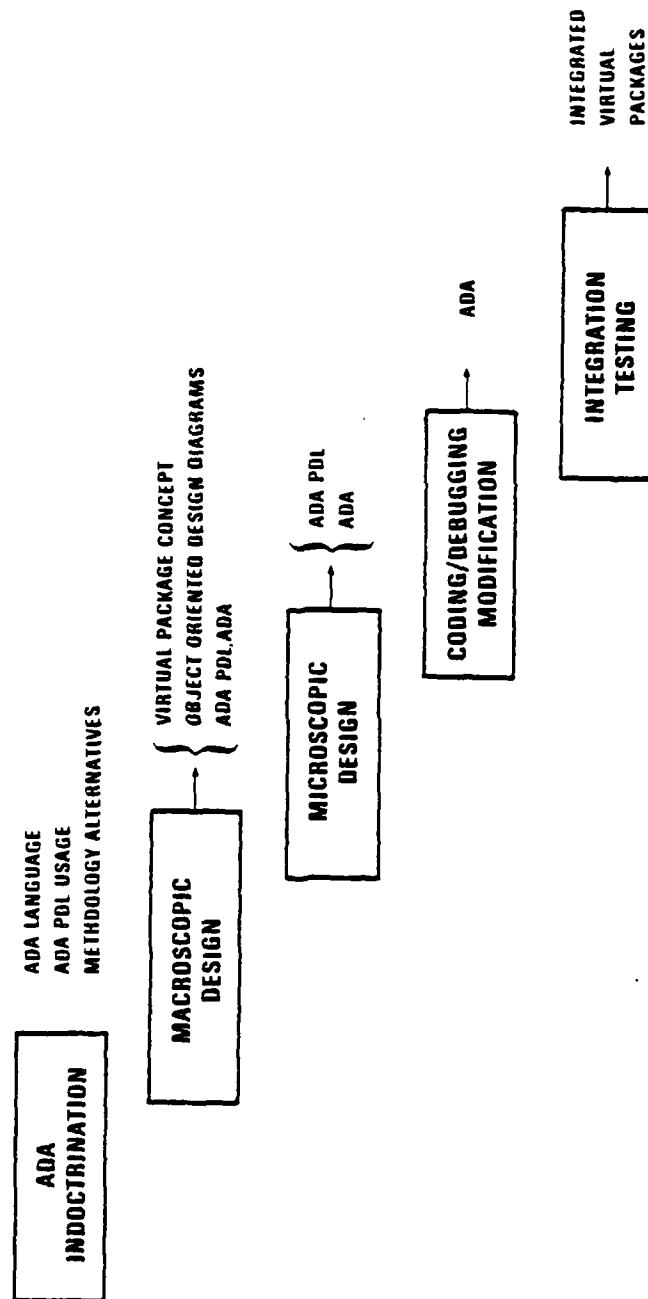


Figure 3.3-1. Software Development Phases

3.2.4 Code/Debug Phase

The last phase in the stepwise refinement process will be to convert the compilation units, both library and secondary, into valid Ada code by supplying the necessary details.

3.3 SOFTWARE DEVELOPMENT AND PROJECT MANAGEMENT

This section describes the overall approach taken with respect to the development of the software and the overall management of the project.

3.3.1 Software Development Phases

The formal software development phases are illustrated in Figure 3.3-1. Not shown is a Development/Performance Evaluation phase which is unique to this project.

3.3.1.1 Ada Indoctrination Phase

The project is initiated with the Ada indoctrination phase. This phase reviews the more novel features of the language to gain more complete understanding of the features and their use. Issues relating to complex features and the use of Ada as a PDL will be explored. Methodology issues and objectives will be reviewed to assure that a proper focus is maintained with respect to using Ada and that the methodology is Ada oriented to the maximum extent possible.

3.3.1.2 Macroscopic Design Phase

The macroscopic design phase will consist of identifying the virtual packages, converting them into the object oriented design diagrams, converting them into the corresponding (refined and extended) PDL, and compiling the Ada library units.

compilation units and subunits, and their dependencies), the definition of formal input/output parameters, and the definition of inputs, outputs, and global data. Major decisions within a module may be indicated as a means of delineating overall control flow. To accomplish the macroscopic design, an attempt will be made to use a proper subset of Ada constructs as a PDL, the objective being to produce compilable modules. This allows an early, increased understanding of Ada without considerable detail, and orients the designs to the language features.

End products of this step will be the formal object oriented design diagrams (OODD), virtually complete library units, which make up the virtual packages, and the PDL designs for the visible subprograms and tasks. As an adjunct, an extracted design summary will be produced which provides compilation unit summaries, call graphs, and other design summary information.

3.2.3 Detailed Design Phase

The detailed design phase consists of applying the next level of detail to the previously completed compilation unit PDL designs and completing the as-yet unspecified private (internal) portions of package bodies.

This design phase is referred to as the microscopic design for the remainder of this document. The microscopic design level of detail will include the definition of the components of the abstract data types, the refinement of all global or common data objects (as opposed to strictly local), assignment of preset and default object values, and the specification of all major control decisions within each compilation unit. The objective will be to produce compilable modules.

During this phase composite test specifications will be produced which define the tests to be performed in debugging and integrating the software.

Table 3.4-4. Specific Performance Requirements

Bit/byte access and manipulation
Provide for insertion of assembly language code
Provide access to operating system functions,
primitives
Provide access to and control of interrupts
Provide access to real-time clocks and associated
timers
Provide macro definition and generation*
Provide generation of I/O tables
Support software modularity
Provide parallel processing constructs
Provide strong data typing
Provide support structured programming concepts
Provide data and control encapsulation*
Provide for formal verification of source code*

* specifically required for trusted software

Table 3.4-5. General Performance Requirements

Provide very high performance
Provide capability to interface with
and manipulate specialized hardware
Provide high transportability of source code
Provide sophisticated data structures
Provide sophisticated control structures
Provide very high reliability

3.4.3.2 Trusted Software Application Requirements

It appears that no studies have been performed which explicitly identify a set of requirements that a language should possess for implementing trusted software. Upon examination of the application area, however, it is apparent that many desirable features are similar or identical to communications applications. Two other features are believed to be strongly related to the characteristics inherent in trusted software. The first is data and control encapsulation. With this ability it should be

possible to construct more secure data and control structures which can be used effectively but without knowledge of the details of the implementation and, therefore, without the ability for unauthorized alteration or manipulation of the structures. The second is formal verification (proofs of correctness) of the designs and source code. Although this evaluation of Ada will not include formal verification of the trusted software source code, indications are that there is a strong correlation between the style in which programs are written and the ability to formally verify those programs /SRII78/.

There is a correlation between the style in which programs are written and the features provided by a language which encourages the writing of programs in a clear, intelligible, and verifiable style or at least proscribes certain undesirable styles. An analysis of the trusted software and Ada features will be made with respect to style and formal verification to determine if any administrative Ada language restrictions are required. Specific criteria to be used for evaluation are Maintainability, Testability, Correctness, Integrity, Reliability, Robustness, formal verifiability of the designs and code, and the ability to retain a usable Ada subset in the context of restrictions which may be required.

3.4.4 Ada Language Characteristics

The last component of overall software quality to be evaluated is the Ada language and its required run-time support environment. Individual features and combinations of features will be evaluated given both the context of the two applications and the interactions with the run-time support environment and the host operating system.

3.5 DATA COLLECTION

This section identifies the sources and data to be collected during the project including the criteria for data collection, analysis and evaluation.

3.5.1 Criteria For Data Collection And Evaluation

The major sources of data and information which will be used to conduct the evaluation are shown in Figure 3.5-1.

The primary objective is the evaluation of the Ada programming language with regard to effectiveness in expressing the designs and implementations and with regard to efficiency of execution. To achieve this objective, several different types of data must be collected and evaluated. Data will be collected regarding which Ada features were used, whether they were used effectively, and whether the resultant designs and implementations were efficient with respect to execution characteristics.

Because of the expressive power of Ada, many different types of software architectures can be formed for any given application. Depending on the architectures selected for implementation, the resulting software may have many different characteristics. Data will be collected and evaluated regarding the specific software architectures that have been formed to determine whether the architectures are suitable for the application and whether they have been effectively represented using Ada.

Since the methodology involving the macroscopic and microscopic design levels was formed specifically for the project, it will be evaluated to determine whether it provides suitable levels of design refinement, whether the information required at each level is necessary and sufficient, and whether the levels are "uniform" with regard to providing a smooth transition from the requirements to the implementation.

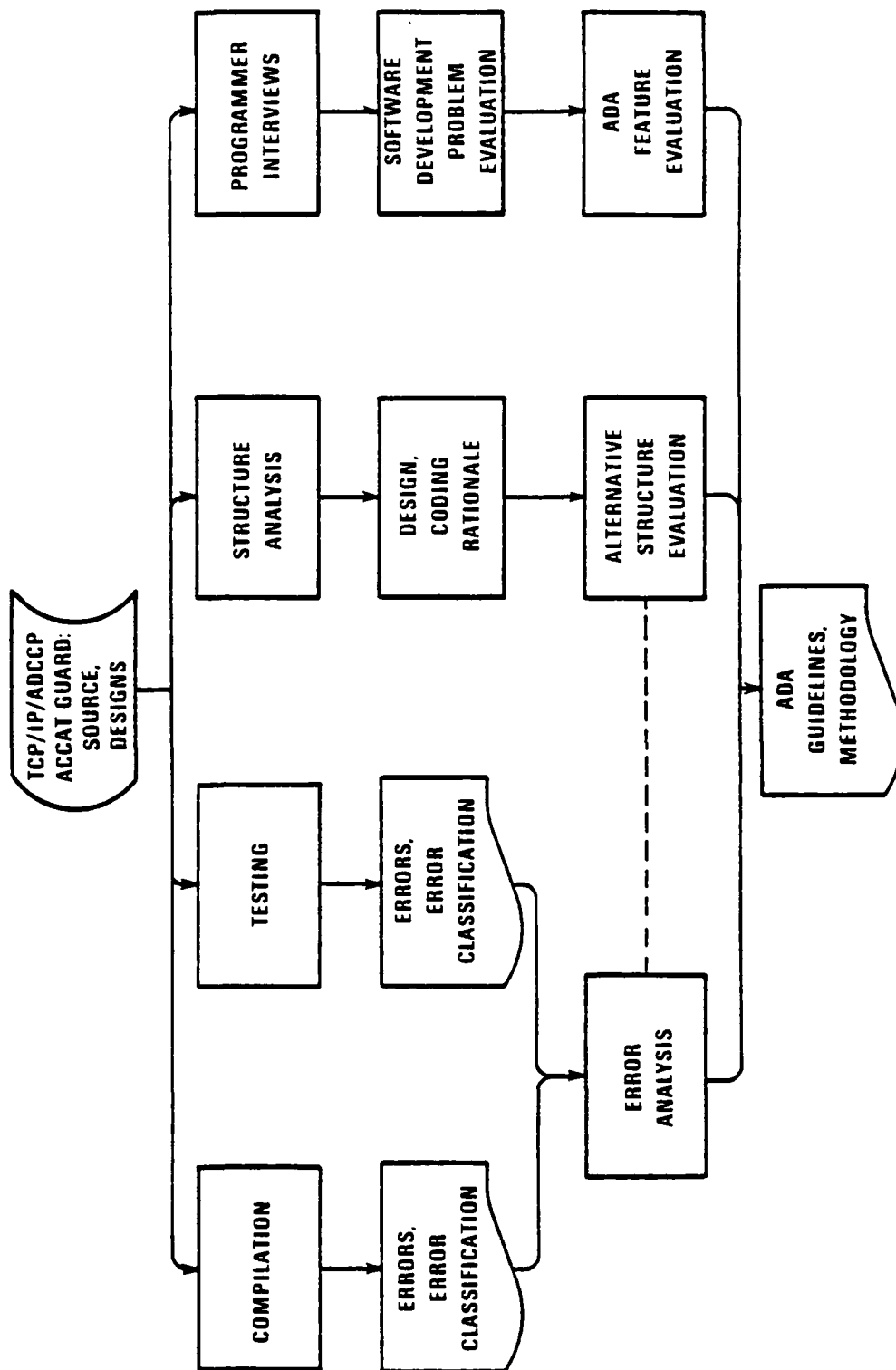


Figure 3.5-1. Ada, Methodology and Architecture Evaluation Components

It is envisioned that there will be a significant interplay between the software architecture, the features of the Ada language, and the software methodology components. A consequence is that various types of errors may be encountered throughout the development process. Overly complex debugging and system integration requirements could result in various types of errors remaining hidden after integration and testing. It is also possible that sequential processing concepts may be applied to Ada tasks with anomalous results produced. To address the interplay among these components, various type of error data will be collected and evaluated.

Although the above areas are of primary concern, incidental evaluation will also be made with respect to: 1) Ada education, both methodology and the language, 2) effects of an Ada Programming Support Environment and a suitable set of Ada-oriented software tools on the software development process, and 3) the characteristics of the respective Kernel Ada Programming Support Environments and their effect on run-time performance and characteristics of the two applications.

3.5.2 Software Architecture Data

The primary objective of the software structure analysis is to determine which Ada features were used and to assess the degree of success or difficulty encountered in their use. The secondary objective is to assess in a qualitative and, if possible, quantitative manner the effectiveness and suitability of the features used. To accomplish the first objective, the software will be examined at two levels. The first level will address the overall organization of the software into modules comprising packages, subprograms, tasks and compilation units and subunits. This organization will be compared with the totality of Ada features and with the software quality factors in order to determine how "good" or suitable the structure is. The second level will address the internal organization of the data structures and bodies of the various modules to assess the

breadth of the Ada features used and to determine the overall composition of the features used. Of particular concern will be whether full advantage was taken of the Ada features or whether a subset of Ada features was used in the style of some other language.

To accomplish the second objective, the Ada features used within each module will be analyzed. In cases where a particular Ada feature, construct, or set of constructs appears to be suboptimal regarding efficient representation in Ada, or especially difficult to implement or understand, a detailed review of the constructs will be made with a view toward finding alternate, improved representations.

3.5.2.1 Software System Architecture

The software system architecture will be evaluated at two sublevels. The first deals with the number, type and interconnectivity of the virtual package architecture. The virtual packages will be evaluated with regard to the Flexibility, Interoperability, Maintainability, Reusability and Transportability software quality factors.

The same software quality factors will be applied to the intra-virtual package architecture. They will be applied to the architecture of the library units which constitute each virtual package. The emphasis will be on assessing the architecture of the software in terms of high-level Ada entities such as library units and the visible components of library units through which control and data flow interactions will be effected.

3.5.2.2 Compilation Unit Architecture

The objective of the compilation unit architecture evaluation is to determine how the visible and non-visible components of each compilation unit are organized from logical and lexical aspects. The emphasis will be on the number, type and interaction of

visible, private, and hidden components and external dependencies. Software quality factors from the development and performance subcategories will be derived from the respective software quality criteria.

3.5.2.3 Compilation Unit Statement Characteristics

The Ada Statement Analyzer will be used to determine the characteristics in terms of Ada source code statement types used. This statement summary will be used as an indication of overall module complexity. Compilation units which are deemed unusually complex or to have other idiosyncrasies will be selected for further review.

Once appropriate compilation units have been identified, the code will be reviewed to determine if problems exist. The software quality factors will be applied for both the software development and software performance categories. The deficiencies, if any, will then be categorized according to the relevant software quality criteria. Selected Compilation Units (CU) will be evaluated to determine the effect of the established programming guidelines on the architecture and the components of the CU.

3.5.2.4 Application-Dependent Architecture Characteristics

The objective of the software architecture data collection is to determine the effectiveness of the system software architecture with respect to external, conceptual models or objectives which were used as primary drivers in shaping the software system architecture designs.

In the communications protocols application the ISO Open Systems Interconnection Reference Model served as the primary architectural model. An adjunct to the functional sublayers is the concept of systems management functions which provide services common to more than one layer. The objectives of data collection and evaluation in this area will be to determine

possible alternative software system architectures and their suitability and to examine how effectively the use of Ada entities was able to capture the system architecture.

Two aspects of the ACCAT GUARD software architecture will be examined. The first deals with the ability to capture the essence of the SPECIAL specifications for the Upgrade Trusted Process and the Downgrade Trusted Process in an Ada implementation. The second deals with the ability to separate trusted and non-trusted software in a manner that prohibits or minimizes data flow from the trusted to the nontrusted components. Special attention will be given to separating KSOS emulation idiosyncrasies from other trusted software problems. Since the ACCAT GUARD application was implemented as a single process in a multitasking environment, an evaluation will be made of the use of Ada tasks for interprocess communication as opposed to UNIX-like ports.

3.5.3 Software Error Data

The overall approach for using software architecture data and software error information and correlating these with the Ada features used is shown in Figure 3.5-2.

The error statistics to be collected comprise two groups: compilation-related errors and execution-related errors. The objectives are: (1) to determine if there are any particular Ada constructs which seem to be systematically difficult to use; (2) to determine which type(s) of errors, if any, remain hidden following a successful compilation and must be detected during execution; (3) to relate errors to module complexity; and (4) to help in the identification of guidelines and alternatives which will either diminish or remove the most severe problem-causing areas.

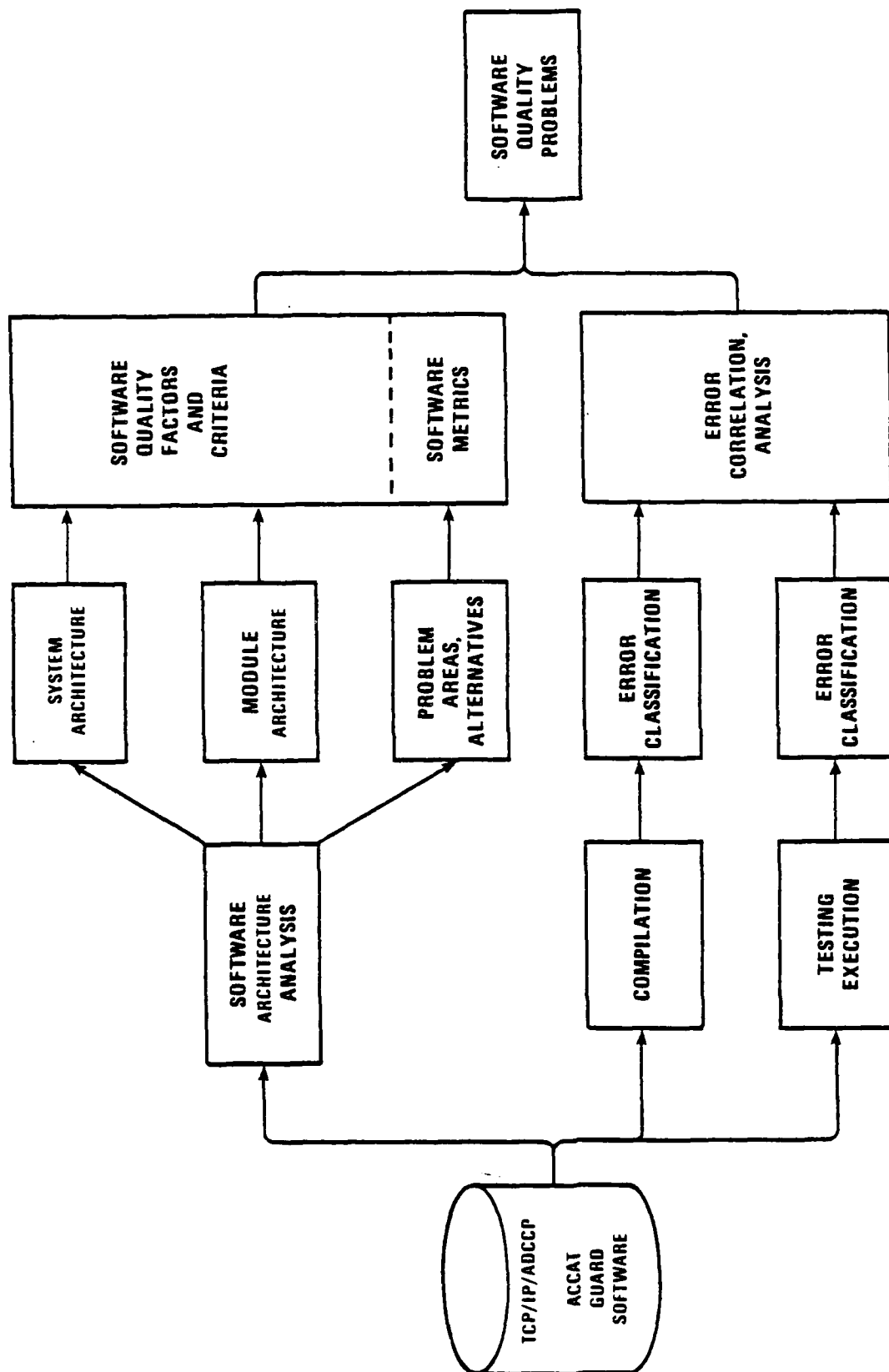


Figure 3.5-2. Software Structure/Error Analysis

SC15967U

Errors encountered for compilation unit or subunit compilation will be identified by type and frequency of occurrence. Execution-related errors detected via unanticipated exceptions, elaboration errors, and erroneous (inaccurate, incomplete, inconsistent) computational results will be grouped by type and frequency of occurrence.

3.5.3.1 Ada Language Errors

Ada language-related errors will be divided into five categories, which are conceptual, syntactic, semantic, execution, and integration errors.

Conceptual errors will deal with Ada language-related problems in which the basic concept of the Ada entity or the purpose of the entity was fundamentally misunderstood and misapplied. Syntactic errors will deal with problems which are encountered in forming the various Ada constructs. Semantic errors will deal with problems which are encountered either during compilation or during execution. The compilation errors will reflect inconsistent use of Ada-defined or user-defined entities which are detected at compile time. Execution-related semantic errors will reflect programs which execute correctly, but produce an unintended result. This type of problem can result, for example, from variable overloading and inappropriate scope declaration or variable identifier qualification. Execution errors consist of those errors which are encountered during either elaboration or execution of program units.

In some instances, errors may not become evident until a significant number of components have been integrated and executed together. This is particularly true of software architectures involving tasking and the dynamic instantiation of generic units. During the system integration process, an "error" log will be maintained by each programmer which will indicate the integration errors encountered and the characteristics of each.

3.5.3.2 Design Errors

Within the Ada programming language a class of errors which can result in an "erroneous program" are identified. These errors result from taking advantage of known facts, such as operating system or run-time dependencies, which lie outside the specification of the Ada language. As a result, the program may work properly in one environment but not in another, or the program may no longer work after an operating system change has been implemented.

Another class of design errors which result in erroneous programs are those implementations which covertly or inadvertently rely on implementation choices made by the compiler or run-time software implementor.

3.5.4 Programmer Interview Data

Interviews will be conducted with the communications protocol and trusted software programmers following the PDR and CDR. The objective will be to elicit qualitative information regarding Ada. Information will be obtained by having one programmer implement a small portion of the other's design as a means of assessing maintainability issues. An attempt will be made to understand the rationale applied in the design and development phase for those approaches which worked, as well as those approaches which had problems. An additional result of this understanding should be the ability to formulate new and improved approaches to design and implementation using Ada.

All project personnel have been directed to maintain a notebook consisting of perceptions, problems, solutions, insights and other useful information which they acquire as the project progresses.

3.5.4.1 Ada Language

Each programmer will be asked to identify those Ada features which contributed most significantly to the respective software development effort and to supply the rationale as to why those features were significant.

Each programmer will be asked to identify deficiencies or weaknesses within the Ada language and to supply the rationale as to what the specific difficulties were. These deficiencies will be divided into those which exist within the language itself and those which are caused by the omission of a specific feature from the language. Those deficiencies within the language will be divided into two groups which address either individual syntactical/semantical forms or problems which exist as a result of combining several syntactical forms.

Those Ada features which are unused will be identified and the reasons for their non-use will be documented.

Certain complex Ada features may be misinterpreted. In those instances, features will be identified and the reason for the misinterpretation will be sought.

A preliminary set of programming guidelines was formed to avoid certain obvious problems in the use of the Ada language, such as excessive nesting of programs or the use of an inordinate number of formal arguments in a single subprogram or task entry call. These guidelines will be reviewed for relevance according to the following criteria: 1-used, effective; 2-used, ineffective; 3-not needed, not used; 4-needed, not used; 5-new addition; 6-change to existing guideline.

3.5.4.2 Methodology

The methodology which is used will be evaluated in these respective areas: the macroscopic design phase, the microscopic

design phase, the code/debug phase, the system integration phase, the relevance of the original design guidelines, and the relevance of software tools for assisting the methodology.

3.5.4.2.1 Macroscopic Design Phase

The macroscopic design phase results in the object oriented design diagrams (OODDs) and the corresponding PDL. Data collection will focus on determining whether the OODD diagrams provide a reasonable initial step for producing a design, how the diagrams should be correlated/coordinated with the higher level preceding requirements, when the OODD should be initiated, what level of detail should be required, and what problems were encountered when using the OODDs. Recommended changes or alternative approaches will be solicited.

Data will be collected on the utility of the virtual package concept with respect to the following criteria: 1) ability to establish early Ada orientation, 2) effectiveness in avoiding inordinate detail at the software architecture level, 3) the ability to provide for enhanced design visibility and understanding, 4) the ability to provide for continuity across software development phases, 5) the ability to provide a basis for software configuration management, 6) the ability to assist in the effective scheduling of software development implementation orders, 7) the ability to address and emphasize software quality factors by indicating the key elements of the designs, and 8) the ability to provide for early incorporation of conceptual models which should influence or guide the overall software architecture design.

The PDL will be evaluated to determine: 1) its correspondence with the OODD, 2) if the appropriate level of refinements have been made, 3) if the PDL is uniform in detail across all units, and 4) what difficulties or problems arise in making the transition from the OODD to the PDL. The PDL itself will be

examined to determine whether it provides the minimum Ada criteria given in Table 3.5-1.

Table 3.5-1. Macroscopic Ada PDL Criteria

Virtual Package Definitions
Compilation Unit Definition
Packages Definition
Subprograms Definition
Task Definition
Formal Parameter Definition
Abstract Data Type Definition
Exception Definition
Major Logic Definition

Another factor which will be evaluated is whether the combination of OODD and PDL are at the suitable levels of abstraction for communicating the information required of a Preliminary Design Review or similar type of review.

As for the macroscopic designs, the number, quality, placement, suitability and uniformity of PDL statements and text contained in square brackets ([]) will be carefully evaluated since this information, along with regular Ada comments constitutes the basis for forming a well-structured design.

Information regarding weaknesses in the use of the OODD and corresponding PDL will also be sought and any changes or recommendations will be incorporated. Careful consideration will be given to potential problems or deficiencies which might result when the approach is applied to software development projects involving larger numbers of personnel.

3.5.4.2.2 Microscopic Design Phase

The microscopic PDL level will be assessed for uniformity of design detail across all modules. Several other factors which are unique to this level of design will also be considered. These include whether sufficient detail and organization existed

generic entities are identified early. They must be completed early in order not to impact other dependent units. The problem of completeness and uniformity of the designs again occurs with respect to proceeding to the code/debug phase. This is very critical here since the only remaining opportunity to rectify omissions is in the code/debug phase.

An issue with the microscopic designs is, again, how to determine completeness. A particular problem with the macro designs of the communications protocols software prior to concluding the microscopic designs is that many of the calls to known entities within the respective bodies were not indicated. Although this problem was easily corrected, the apparent cause appeared to be one of oversight and of not having a specific requirement to do so in the methodology.

4.1.4 Code/Debug Phase

No major problems were found with the procedure of converting the microscopic designs into Ada code. In the communications protocol software, considerably more design was required to complete the coding. One consequence of this was that the code/debug took longer than anticipated. Another significant problem occurred related to the Ada compiler. The implementations had to deviate from some designs because certain Ada features were not implemented and compiler-related errors caused considerable numbers of workarounds to be attempted to obtain working code. Work on the trusted software application was suspended for approximately six months while awaiting delivery of a validated compiler. Several compiler problems were traced to heap overflow and heap management problems, resulting in tasking that frequently did not work properly. In part, these problems were solved by relocating code from the bodies of tasks to their containing package bodies and forming subprograms of the relocated code to reduce the size of tasks.

suggested. This resulted in a considerable number of problems which could easily have been detected during the macro phase instead of the later micro and code phases. As a result, the design guidelines have been changed making compilation of at least the package specifications mandatory for completion of each of the design phases.

4.1.3 Microscopic Design Phase

At the beginning of the microscopic design phase the virtual package structure, the compilation unit structure, and the corresponding PDL of the macroscopic design phase should be well defined and ready for further refinement. A significant step which can contribute to assuring this is to perform some portions of the microscopic design prior to the formal conclusion of the macroscopic design phase. This has the benefit of uncovering lower level details which may influence the overall compilation unit architecture or the allocation of capabilities across the various compilation units.

During this phase the secondary program units were more completely defined by supplying the necessary bodies. Tertiary and lower level program units were also defined. PDL statements which were qualitatively similar to the macroscopic PDL statements were supplied. In the case of primary routines, the previous PDL statements were expanded into more detailed PDL statements and into direct Ada code in some instances. The declaration of local data types, objects and default values was completed. By the completion of the microscopic design phase, the primary program elements were essentially completed except for minor processing details.

It was recognized that generic declarations needed to have their bodies essentially completed since they would be some of the first entities used to produce other nongeneric entities. An important factor here, and in the overall methodology, is to assure that

4.1.2.3 Macroscopic PDL

Since it is neither possible nor desirable to communicate all the design information such as completed types, formal parameters, guard conditions and similar entities via graphical representations, the macroscopic design phase was augmented with the use of Ada as a PDL. The objective was to provide a means of refining or extending the OODDs by incorporating additional detail that advance the designs one more step into Ada code. The refinements or extensions to be included were: completion of exported data types and objects, default value assignments, declaration of all formal parameters for subprograms, task entries and generic declarations and instantiations, declaration of exceptions, declaration of blocks, declaration of nested secondary subprograms or tasks, specification of major logic decisions in the visible components of library units, at the very least those exported from the virtual package, and the use of PDL comments (text included in square brackets) which would ultimately be converted into bona fide Ada code.

In general this approach worked very well, but there were some notable problems. Names of called subprograms or task entries were either omitted or not clearly indicated even though the entities were defined. The incorporation of exceptions was omitted throughout the macroscopic designs even though there were numerous instances where they could occur and should be managed. The most difficult problem was knowing when the designs were completed, that is, when all the necessary information had been supplied. This is significant since providing too little detail will result in requiring top-level design to be performed at the microscopic level, while providing too much detail will generally result in uneven levels of detail because of schedule considerations.

During the early portions of the macroscopic design phase in both applications, the package specifications were not compiled as was

on Ada-based graphical design representations become known to the project. The first was the Object Oriented Design approach of /BOOC83/ and the other, in preliminary form, was presented at the February 1983 San Diego AdaTEC meeting and is now documented in /BUHR84/. A synthesis of these approaches was made with refinements and extensions to provide representations that would satisfy the project needs.

The manual creation, modification and control of these diagrams during the initial portion of the project was barely tractable. During the design process, much iteration and refinement occurs until all compilation units are defined and the visible components are defined and properly distributed. This process is even more complicated when tasking interactions are required to achieve resource sharing/protection and inter-protocol-layer communication.

A second problem was determining which Ada entities should be represented since different entities communicated different levels of information. For example, if only task specifications were represented, it would not be possible to indicate that entries were guarded, if that were the case, since guard conditions can be represented only in task bodies. Similar considerations prevail for timed/conditional entry calls, selective waits, discriminating between subprograms which are functions and those which are procedures, and distinguishing between units which are generic declarations and those which are generic instantiations. This has been resolved by permitting the maximum level of architectural information to be communicated. This is consistent with providing the necessary information required to conduct a preliminary design review and indicate what the key software architecture considerations and factors are and thus provide the necessary design visibility at an early stage of software design or possibly in precursor form prior to concluding the requirements phase.

these problems have been resolved and the virtual package concept has been refined and is an integral part of the methodology.

During development, it was recognized that virtual packages could be related directly to Computer Software Components (CSCs) of the DOD-STD-SDS. The formation of virtual packages during the requirements phase would provide an excellent opportunity for iteration between the requirements and design phases. This feature is critical since it frequently is the case that requirements can be optimally organized and more clearly stated if some preliminary design is performed prior to requirements definition. A note of caution here is that care must be taken to assure that Ada language characteristics do not unduly influence requirements. In cases where CSCs were large, or design criteria such as reusability or transportability were factors, it was also the case that requirements could be preserved and that nested virtual packages could be used.

An additional benefit of virtual packages is that a designer can organize Ada packages into those visible and those hidden from other system components (virtual packages). The virtual package has many of the characteristics of an Ada package except that it is not necessarily a compilable entity. This has the side benefit that software architectures other than nested architectures can be cleanly represented while achieving design visibility and modularization.

4.1.2.2 Object Oriented Design Diagrams

Since virtual packages partition the system into top-level "Ada" units, the next step is to begin stepwise refinement. It was determined that only information which could be conveyed via Ada library units should be represented within each virtual package as a way of indicating the architecture of that virtual package, and that this information should be conveyed graphically to minimize premature attention to details. Two significant works

abstraction were selected. These are the macroscopic and microscopic design phases which were followed by the code/debug phase and the system integration phase. In completing the design philosophy, one additional problem was recognized: at the early stages of design, the package concept represented too fine a level of detail and the program library represented too coarse a level of detail. Although the Ada language does permit the use of sublibraries or their equivalent, it does not require sublibraries. To deal with this granularity problem, the virtual package concept was formed and implemented.

4.1.2 Macroscopic Design Phase

This section presents the analysis of the use and effectiveness of the elements of the macroscopic design methodology.

4.1.2.1 Virtual Package Concept

The virtual package is designed to show the first level of "Ada" design in progressing from the requirements definition to design and to improve continuity across the requirements-design boundary. The concept emphasizes the top-level architectural components in terms of Ada library units and avoids premature detail.

In both applications, the virtual package concept was used as the point of departure for producing the design from the very high level requirements contained in the original proposal and the specifications supplied. In using the virtual package approach, several additional considerations had to be addressed. Early versions were unclear as to marking imported and exported components, the level of intermodule interconnection which should be achieved, how best to annotate the diagrams, whether virtual packages containing a single Ada package were acceptable, how to deal with generic units, and nesting of virtual packages. All

SECTION 4

ANALYSIS

4.1 SOFTWARE DEVELOPMENT METHODOLOGY ANALYSIS

This section presents the analysis of the design methodology which was used for producing the designs initially and for revising the designs during the transition from SIP/ADCCP to TCP/IP/ADCCP.

4.1.1 Overview

An overall goal of this project was to develop an Ada-based design methodology for the types of applications under consideration. In analyzing this goal, several objectives were established which are identified in Table 4.1-1.

Table 4.1-1 Software Design Methodology Objectives

- o Provide for early Ada orientation of design
- o Avoid premature Ada details
- o Provide for early software architecture design visibility
- o Provide for detailed software design visibility
- o Provide for design continuity across development phases
- o Provide basis for configuration management
- o Use Ada features to support software engineering principles
- o Incorporate existing models and architectural principles

To use Ada most effectively, the methodology should be Ada-based with Ada selected for use as the PDL. To achieve the early Ada orientation and to minimize premature involvement in Ada details, the graphical representations of the object-oriented design methodology of /BOOCH83/ were selected to precede the use of Ada. Since software design proceeds best via stepwise refinement and can frequently be organized around the overall architecture and the lower level detailed designs, two levels of design

3.5.5.3 Programming Support Environment Issues

Because of the underlying complexity of Ada and the fact that the distinction between operating system functions and run-time support functions has been made less precise, it is likely that various programming support environment or run-time support characteristics, which are beyond the direct control of the programmer, will impinge on the designs and consequently on the performance. The objective here is to collect and note such relevant data.

An additional set of criteria used for evaluation of the trusted software are the Class A1 - Verified Design Criteria for Trusted Software of /USDO83)/. Primary elements of evaluation in this area are identified in Table 3.5-3.

3.5.5.1.3 General Performance Considerations

Several criteria affect the Correctness, Efficiency II, Integrity, Reliability and Robustness software quality factors which are common to both applications. These criteria are identified in Table 3.5-1.

Table 3.5-3. Class A1 - Verified Design Criteria

Object Reuse (4.1.1.2)
System Architecture (4.1.3.1.1)
Covert Channels (4.1.3.1.3)
Security Testing (4.1.3.2.1)
Design Specification & Verification (4.1.3.2.2)
Design Documentation (4.1.4.4)

3.5.5.2 Ada Language Issues

The purpose will be to review the existing implementation and determine if alternative implementations could have been used which conceivably would have better performance characteristics with respect to any of the software quality factors. To the extent that such alternatives can be identified and project time permits, alternative implementations may be attempted to determine if performance can actually be improved.

Table 3.5-2. Software Performance Criteria

SOFTWARE TESTS

TCP/IP/ADCCP

FUNCTIONALITY TESTING:

TCP

- MISSING SEGMENT(S)
- DUPLICATE SEGMENT(S)
- SEGMENT CHECKSUM ERRORS
- SECURITY/PRECEDENCE VIOLATIONS

IP

- DATAGRAM CHECKSUM
- DESTINATION-UNREACHABLE
- TIME-TO-LIVE
- INVALID-SUBNET-PARAMETERS

ADCCP

- OUT-OF-CONTEXT COMMANDS
- OUT-OF-CONTEXT RESPONSES
- TIMEOUTS
- INVALID FRAME ERRORS
- CRC ERRORS

LINE CONTROL MODULE (LCM)

- TIME-OUTS (LINE DROP)
- DATA ERRORS (BIT DROP)

ACCAT GUARD

FUNCTIONALITY TESTING:

- HIGH-LOW MAIL
- LOW-HIGH MAIL
- HIGH-LOW QUERY
- LOW-HIGH RESPONSE
- LOW-HIGH QUERY
- HIGH-LOW RESPONSE
- DOWNGRADE REJECTION
- HIGH/LOW BUFFER WATERMARKS
- GUARD TERMINATION
- DOWN GRADING
- SANITIZATION

ADA-SPECIFIC EFFICIENCY II CRITERIA

PRAGMAS: CONTROLLED, INLINE, OPTIMIZE, PRIORITY, SHARED, SUPPRESS

TYPES/OBJECTS: DYNAMICALLY VS. STATICALLY CREATED OBJECTS

SUBPROGRAMS: EFFECTS OF EXTENSIVE ELABORATION

TASKS: REGULARITY, ACCURACY OF EVENT TIMING, INTERRUPT PROCESSING
TASK ACCESS ALTERNATIVES

EXCEPTIONS: HANDLING, PROPAGATION, TASKING INTERACTIONS

GENERIC: EFFECTS OF DYNAMIC INSTANTIATIONS

IMPLEMENTATION-DEPENDENT FEATURES: UNCHECKED PROGRAMMING

3.5.5 Software Performance Data

Software performance data will be collected from the respective application requirements and characteristics, Ada language issues, and programming support environment issues.

3.5.5.1 Application Architecture

Application-dependent requirements will be identified, and the performance of the software will be evaluated with respect to those criteria.

3.5.5.1.1 Communications Protocols

The communications protocol performance tests relating to protocol error conditions of Table 3.5-2 will be generated and used to evaluate the Efficiency-II criteria. Of specific concern is whether the completed program contains errors relating to performance and, if so, the source of these errors.

3.5.5.1.2 Trusted Software

Because of the emphasis on correctness in trusted software, the trusted software application will be evaluated very carefully with respect to Maintainability, Testability, Correctness, Integrity, Reliability and Robustness. Within these software quality factors, specific attention will be given to the spurious occurrence of exceptions, their handling, and their impact on the overall software. These conditions will be tested by generating the maximum number of messages possible and verifying that the first, last or requeued messages do not become lost or erroneously transferred and deregistered. The types of operations which will be used for evaluating the overall performance of the trusted software are presented in Table 3.5-2.

3.5.4.3.1 Communications Protocols

The original basis for the design was the top-level network architectural design and the low-level protocol specifications for SIP and ADCCP. A set of "requirements" was formed which enabled the relevant components of the architecture to be used to produce a mini-network. This is substantially different from the one described for the trusted software application.

A transition to the TCP/IP protocols and a revision of the ADCCP protocol was made prior to the completion of the macroscopic and microscopic designs for the SIP/ADCCP protocols. This shift in requirements prior to the completion of the original designs presented an opportunity to assess the methodology with respect to changing requirements and design flexibility. Specific factors of the methodology to be evaluated are the ability to use a partially completed design as a reference for incorporating a new set of requirements and to isolate portions of the design requiring modification.

3.5.4.3.2 Trusted Software

Both requirements and design documentation (/WOOD781, /LOGI79B/, /BALD79/) were in existence for the entire ACCAT GUARD application. Also available was a preliminary draft of the Upgrade and Downgrade Trusted Processes /LOGI79A/ and the adjunct software that was used as a basis for the trusted software design. The transition to Ada was complicated by the fact that the trusted processes were written in SPECIAL and that the basic design and requirements were couched in UNIX terms which required translation to Ada.

3.5.4.2.5 Design Guidelines

A preliminary set of design guidelines was formed to assure that a certain minimum set of information was provided at each level of design. These guidelines will be reviewed for relevance according to the following criteria: 1-used, effective; 2-used, ineffective; 3-not needed, not used; 4-needed, not used; 5-new addition; 6-change to existing guideline.

3.5.4.2.6 Software Tools

During this project no software tools were available, other than text editors and two Ada compilers, until just prior to the completion of the Draft Detail Designs for the TCP/IP implementations. At that time SKETCHER, an on-line, interactive graphics support tool, became available and was used to produce the object-oriented design diagrams. Since software tools can contribute significantly to making a software development methodology effective and efficient, a list of desired tools and their functionality will be compiled.

3.5.4.3 Project/Application Evaluation: Alternatives/Retrospectives

The two application areas were implemented from substantially different starting points and with different requirements. To assess the impact of individual characteristics on the software development methodologies, "free form" information will be collected. The objective here is to evaluate the effects of different baseline information on the design methodology and the use of Ada.

at the macroscopic level in order to proceed directly with the microscopic design or whether substantial changes had to be made to the macroscopic design before proceeding and, if so, what the causes were. Other design related problems will be documented. Since this level of design requires the fundamental completion of the macroscopic designs in greater detail as given in the design guidelines, the level of completion will be evaluated and the suitability of the microscopic design, including the secondary and tertiary modules, will be evaluated. The number, quality, placement, suitability and uniformity of PDL statements and text contained in square brackets ([]) will be carefully evaluated since this information, along with regular Ada comments, constitutes the basis for forming a well-structured design.

3.5.4.2.3 Code/Debug Phase

Two key factors to be evaluated are the degree of design changes, extensions or variations that had to be made, and the underlying reasons. These are significant in that they potentially indicate a deficiency in the design with respect to level of detail provided or to the soundness of the design. In the latter case, problems may be an indication that the methodology may need to be revised to require more or different detail earlier in the software lifecycle.

3.5.4.2.4 System Integration Phase

The system integration phase will be somewhat different in nature than the classical notion of system integration. Segments of code will have been "integrated" through the use of context clauses during the macroscopic and microscopic design phases and the code/debug phases. Traditional problems will have already been addressed. Attention will be given to specific design or performance issues which arise during system integration to determine how they are related to the design methodology and what changes need to be made.

4.1.5 System Integration Phase

This phase has different characteristics from normal system integration due to the separate compilation capabilities of Ada. It is possible to achieve incremental system integration, at least at the library unit interface level, if library units are made "compilable" in the macroscopic and microscopic design phases. The benefits are that typical interface incompatibilities are eliminated early in the development cycle. As a result, emphasis can be placed on more difficult integration issues of the semantics of control and data flow sequences with regard to the visible components of library units.

In large systems which may contain hundreds of packages, the use of the virtual package concept also can help to identify which components need to be available to higher level units in the system. The virtual package concept also aids in formulating top-level schedules early in the design process, reduces schedule difficulties, and assists in producing effective integration schedules.

One software tool which could assist in the system integration phase is an executable PDL. By being able to explore alternative designs and achieve top-level integration in an incremental manner, many design problems could be identified and resolved as the design evolves. The need for early identification and resolution of design problems is especially important in large systems which depend on a multitasking/multiprocessing environment.

4.1.6 Design Guidelines

A small set of design guidelines was formed at the outset of the project. The primary purpose was to assure that some minimal level of design information, consistent with Ada syntax and semantics, was achieved. Although the guidelines generally

proved effective when they were used, they are viewed as being not very comprehensive. The imposition of design guidelines may have other purposes, such as proscribing certain Ada features, depending on the application, to assure transportability or reusability. The result of the guideline evaluation is given in Appendix A.

Ada is a very robust language with the ability to achieve many different implementations for a given set of requirements. When such factors as software architecture, execution efficiency, transportability and maintainability are considered, it is necessary to specify the degree of importance of the software quality factors that each virtual package or package is to have. If this is not done, designs with extensive elaboration requirements may result with a detrimental impact on execution efficiency, especially if the elaborations are performed in frequently called subprograms. Similar effects may result from the use of recursively formulated algorithms and dynamic instantiation of generics.

4.1.7 Programming Guidelines

A general principle in formulating the programming guidelines was that the list should be small and simple. The objective of the guidelines was to assure some level of uniformity across compilation units and to eliminate some of the more obvious problems. Appendix A contains the evaluation of the guidelines and includes additions which have been made as a result of project experiences.

The guidelines were generally effective when used. However, the guidelines were not always used or interpreted correctly. They did not include transportability or reusability goals. This will generally require more effort, especially when tasking and implementation features are considerations. Moreover, the guidelines may well be a function of the application itself such

as determining how to partition the systems management functions with regard to the individual layer functions in a communications node application.

4.1.8 General Design Methodology Factors

This section presents the analysis of several factors which either span design phases or which deal with design issues not specific to a given phase.

4.1.8.1 PDL Characteristics

One topic of continued debate is whether a PDL should be directly compilable or whether the PDL should be Ada-like and perhaps "compilable" via a PDL processor. The advantages of compilation are that a separate PDL processor is not required and more detail will generally be supplied. Specific disadvantages are that the use of TBD-types and TBD-object statements may require considerable effort to include or parameterize from a PDL-standards package, and do not provide additional design information that can be assessed. There is also the risk that designers become the programmers and become enmeshed prematurely in coding details which can obscure top-level architecture design objectives. Where the objective is to indicate control flow, based on requirements, through the use of if and case statements, it may be more desirable to communicate initially what the conditions are, via the use of embedded English language statements, as opposed to the specific variables that are used to form the expression of the condition.

Another factor of an Ada-based PDL is whether the PDL should be executable and precisely what "executable" means. One benefit of having executable PDL is that the semantics of the designs and their dynamic aspects can be verified incrementally as the design proceeds from one level to the next. In sequential types of applications, such as mathematical applications, this appears to

be less important. However, in applications involving tasks, an executable PDL could provide considerable support in achieving early "integration." Complex intertask interactions could be evaluated, and errors corrected early in the design cycle. A specific problem to be resolved is determining the features or attributes of an executable PDL with respect to what specific goals are to be achieved. A simple capability could provide a trace of the control flow given statically determined conditions. This could be expanded to permit conditions to be varied either based on program computations or via some type of operator interaction. Problems dealing with whether or how to use English language statements, if permitted in the PDL, need to be resolved.

A question regarding the development phases is determining when the design of a particular phase has been completed. This can be related to PDL expansion ratios in making the transition from one phase to the next. A procedure can be declared with one Ada statement. If it is assumed that a procedure is limited to approximately 200 Ada statements, then some possible expansion sequences to progress from the declaration through the macro/micro design phases into the code might involve multipliers of approximately 20, 3 and 3, or 20, 5 and 2 or 20, 2 and 5 or 10, 5 and 4. A way of determining when a given level of design has been completed is to sketch the top level design and apply the expansion ratios.

Table 4.1-2 indicates ratios for a limited sampling of modules from both applications and provides composite results. In comparing the regularity of the calculated ratios, one possible interpretation of the high ratio of 3.2 for the HGSD is that perhaps the macro PDL for this module should have been larger than it actually was and that perhaps this was detectable during the design review process.

Table 4.1-2. PDL Expansion Ratios

Module	BUFFMGT	HGSD	LGSD	AVERAGE
Macro	59	33	25	
Ratio	1.3	3.2	1.4	2.3
Micro	76	106	60	
Ratio	2.0	2.0	2.2	2.1
Code	150	215	133	
Composite	2.5	6.5	5.3	4.8

To form a basis for the use of expansion ratios, additional work is needed to determine their validity based on such factors as type of computations, effect of comments, effects of architectures, etc. This could provide useful information that would contribute to assuring that designs were completed at the correct level of detail. Application of expansion ratios could be used to determine the soundness of code estimates by providing an estimate which could be compared with other efforts, an available data base of information, or general assessments by experienced individuals as to the magnitude of the effort.

Another issue is whether the PDL should be maintained with the code or whether the PDL should be maintained separately from the code. Each approach has several advantages and disadvantages. In both applications, two levels of PDL were maintained separately from the code except for the SPECIAL specifications of the trusted process. Advantages of maintaining the PDL and code separately are that source files are smaller, easier to manage, and easier to read; and the question of whether to maintain PDL and code in contiguous blocks or interleaved is avoided. Disadvantages are that crosschecking of any of the entities involves examining two or three separate sets of information. There is also greater chance that PDL levels will not be consistent as development progresses due to design changes. There is the need to explicitly and separately update the documentation after the code has been completed, assuming the PDL will be required for future maintenance efforts.

4.1.8.2 DOD-STD-SDS Compatibility

The development of the methodology was not oriented toward a specific set of documentation, due to the differences in documentation standards across the DOD services and agencies. It was also anticipated that new documentation standards would be formed that were compatible with Ada as an implementation language and with the use of Ada-based PDLs, since many of the existing standards are generally not compatible.

Although this evolution is taking place, another development, namely, the formation of the new (draft) DOS-STD-SDS. It appears the methodology will have to be combined with the final version of DOD-STD-SDS. A small study effort was initiated to gain familiarity with DOD-STD-SDS and assess the compatibility of the methodology. The DOD-STD-SDS documentation requirements, as recommended, are highly compatible with the development methodology. Three different aspects of compatibility, including development phases, components hierarchy and design components, were briefly assessed. In the DOD-STD-SDS nomenclature, the Computer Software Configuration Item (CSCI) and the Computer Software Component (CSC) are simply new names for the previous MIL-STD-490 Computer Program Configuration Item (CPCI) and Computer Program Component (CPC).

Although there is strong compatibility between the two, there are several issues which need to be addressed either universally or on an ad hoc basis when documentation tailoring occurs or specific project requirements are established. One of these involves how the AdaPDL will be used, what data and information are derived from the PDL, how the derived data and information are organized and presented, and where they are presented. Through the use of AdaPDL, it is possible to specify such items as protocol headers with respect to the component names and data types and even the physical data layouts through object and type declarations for records and representation specifications to fix

record component locations. To reduce development and documentation times and schedules, it should be permissible to accept such Ada representation directly in lieu of other formats. Tables of set/used information can be derived from the PDL and readily organized in several different ways to enhance the visibility of the information. Other issues which are presently in conflict with Ada entail the use of overloading, single entry/exit requirements, the restriction on the use of language key words such as `_TYPE` as a suffix and the use of code which is dynamically "self-modifying" code; such as generics which are dynamically instantiated and generic objects which take on values determined by computational results. One final item is the adaptation of documents, such as the Software Detailed Design Document and Software Top-level Design Document, to be compatible with Ada entities such as packages, context clauses, and exceptions. These elements can be readily identified and easily located by reviewing personnel who need not be familiar with the PDL.

A final design issue entails the schedule proportions for the macro and micro design, code/debug and software integration phases. In the original schedule formulations, both applications were treated identically, with proportions of 2 months, 4 months, 3.5 months and 3 months, respectively. The 50% allocation to design is considerably higher than has been typical. The approximation of 25% for integration is considerably lower than the commonly touted 50% figure. Although no specific conclusions can be drawn from these figures because of the many compiler limitations and problems, some specific points can be addressed. Given a software engineering emphasis, it will be required to devote considerably more time to the design phase than there has been in the past. Because of the data abstraction capabilities of Ada, data-oriented design should become more integral to the design process in terms of viewing an algorithm as a combination of both data and control structure. The design time required will increase, both absolutely and relatively, when software

transportability and reusability requirements are imposed on an application. To the extent that truly capable PDL processors and executable PDL processors are available, there will be a justification for reducing the system integration time. If the emphasis is placed on both levels of design, the coding phase can also be smaller than normal, since some units will essentially be coded and the process of converting the designs into code should be rather direct and require little high-level thinking to effect the transition.

4.1.9 Application Dependent Methodology Characteristics

This section addresses various factors which are strictly related to the specific characteristics of one of the applications.

4.1.9.1 Communications Protocols

4.1.9.1.1 Transmission Control Protocol (TCP) and Internet Protocol (IP) Specifications Issues

Following the decision of DCA to terminate the effort on the Segment Interface Protocol (SIP) and make the transition to the TCP and IP protocols, two significant issues arose. The first issue involved the on-line availability of the TCP and IP Specifications; the second involved the interpretation of the Specifications with respect to requirements and design.

Since both specifications contained considerable "design" information and much of this information was organized in quasi Ada Program Design Language format, the question of on-line availability of the specifications arose. The primary motivation was to minimize redundant work and capture as much information as possible. After telephone conversations with Defense Communications Engineering Center personnel, it was determined that the Specifications were not readily available on-line to the SYSCON VAX. The alternative was to re-create the information

needed as part of a new design or to enter the specification with a high probability of recovering a significant portion. Subsequently, the specifications were placed on-line from the textual hardcopies.

In some instances it seemed that a specific design was implied by the Specifications. Specifically, there was a question of the intent of the Specifications regarding the interface between the TCP and IP protocols WITHIN a given machine for a given implementation. The question was whether an actual record format must be exchanged between the two layers or whether an access variable pointing to the record could be exchanged. At issue is not the compatibility of two peer layers in two different machines in two different implementations and two different languages, since that could be achieved as long as the TCP or IP protocols are followed. Rather, the problem is one of protocol-layer software transportability and interoperability since a TCP implementing the TCP/IP interface as a record structure will be incompatible with an IP implementing the interface as an access variable. This lies outside the Ada language capabilities and is not clearly addressed in the IP Specification. Section 10.2 of the IP Specification seems to allow enough latitude to implement the InterProcess Communication (IPC) any way desired. A similar situation exists in the TCP Specification in Section 6.5.4.1 and 6.5.4.3 where the FROM_ULP and TO_ULP data structures are specified and are implied as record structures to be passed as opposed to a parameter list or access variable.

4.1.9.1.2 Transmission Control Protocol (TCP) and Internet Protocol (IP) Transition Issues

Following the completion of the macro/micro designs for the original SIP/ADCCP protocols and following the beginning of the implementation of the application layer and system management layer software, SYSCON was directed to discontinue work on the SIP/ADCCP protocols and begin implementation of the newly

standardized TCP/IP protocols. Since a change in requirements frequently occurs on projects, this presented an ideal opportunity to assess how effective the methodology would be in supporting changes in requirements and design. Several features of the methodology are evaluated below and assessed as to how well they supported making the necessary requirements and design changes.

Figure 4.1-1 presents an overview of the transition. The virtual packages were organized according to the OSI Reference Model architecture which assisted considerably in directly identifying major pieces of the software architecture which needed to be reviewed. The HOST_UNIT_SERVER, HOST_SIP_SERVER, TERMINAL_UNIT_SERVER, TERMINAL_SIP_SERVER, and NETWORK_SIP_SERVER VPs were identified for deletion. The HOST_ADCCP_SERVER and NETWORK_ADCCP_SERVER VPs were identified for modification to incorporate the standard ABM mode of ADCCP as opposed to the AUTODIN-II specific mode. The SYSTEM_INITIALIZATION, COMMUNICATION_CONFIGURATION, SYSTEM_TESTS and SYSTEM_MONITOR VPs were identified as requiring changes to accommodate changes necessitated by the TCP/IP requirements themselves. The next step was to examine the OODDs to determine which compilation units would require alteration in terms of either deletion or modification. The next step was to examine the micro PDL to isolate the changes to particular components of the packages. The PDL was extremely helpful in that it was possible to concentrate on the architectural level of the design rather than the code details. The micro PDL was selected because the same person who performed the initial design was making the design changes. If another person had been making the design changes, it would have been desirable to include the macro level in making the transition.

This type of analysis was carried through to the coded modules. Since the existing designs could be reviewed incrementally, it greatly facilitated identifying areas requiring changes and the

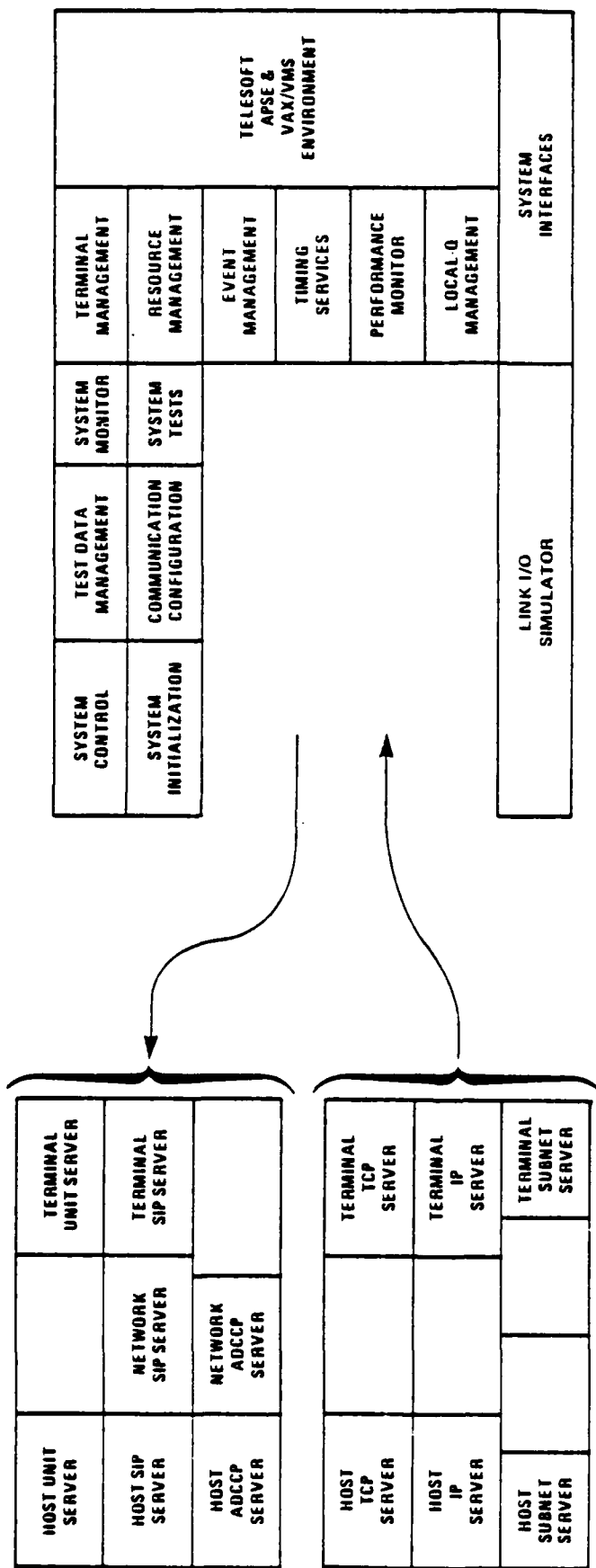


Figure 4.1-1. Communications Protocols Detailed Architecture Transition Components

exact nature of the changes. The SIP macro PDL contained too much detail and required more effort to review than it should have. This resulted in proceeding directly to the micro PDL.

During the code/debug phase another lesson became clear. In systems where complicated tasking occurs, attempts should be made to conduct some amount of prototyping which results in executable skeletons. The benefits are that basic control and data flow paths can be established and evaluated prior to committing effort to refining the PDL.

The macro/micro levels of abstraction, if they contain the correct level of detail, can assist considerably in making modifications to existing designs and code by enabling quicker identification of modules to be retained, deleted, modified and added. By following the macro/micro design for the new requirements, it is possible to retain the overall software architectures and follow the same methodology for the inclusion of the new requirements.

4.1.9.2 Trusted Software

The design of the ACCAT GUARD application was based on /W00D78/, /LOGI79A/, /LOGI79B), /BALD79/. The FGTP and UGTP were formally specified using the language SPECIAL. In producing the macroscopic and microscopic designs, the SPECIAL specifications were included directly as annotated comments in the respective Ada units. Since these specifications were a preliminary version, they contained several errors and oversights which complicated the creation of the Ada PDL. Interpretations had to be made as to what was intended with the result that some requirements which were implied were not captured directly. There is difficulty in translating a computer program from one language to another with respect to syntax and semantics. The case with SPECIAL was further aggravated due to the preliminary nature of the specifications. In trusted software, where

considerable emphasis is placed on correctness, completeness, consistency and clarity, translation is a significant problem. One way of eliminating this problem is to use a language for formal specification which is compatible with or is a dialect/superset/subset of the implementing language. The language ANNA (Annotated Ada) illustrates such a language.

Given that a set of trusted software requirements exists, the issue becomes one of how the methodology can effectively support translating the requirements into a design and implementation. With regard to the macroscopic/microscopic design phases of the methodology, two elements are central. It is necessary to separate the trusted and nontrusted software in order to establish well defined boundaries. It is necessary to further separate the user-visible trusted software from the nonvisible trusted software so that appropriate secure interfaces can be defined. The methodology achieved this separation very well beginning with the virtual package concept. At the macroscopic and the microscopic levels, it is relatively straightforward to include the ANNA as a way of supplying more complete semantic information and providing the formal definition of the design as required by the Formal Top-level Specification.

With regard to translating the nontrusted software from the requirements into the macro and micro designs, there were no basic difficulties encountered.

4.1.10 Software Tools

Throughout the project, only three software tools were used. These consisted of the VAX/VMS screen-oriented editor, EDIT/EDT; the Ada compiler; and a program for converting all Ada keywords to lower case. Prior to the completion of the microscopic designs for the TCP/IP implementation, SKETCHER, a software tool developed under IR&D funding, was completed. SKETCHER is designed to interactively produce OODs in ASCII character format

using a VT-100 type terminal. The need for SKETCHER was a direct result of the work performed on the DCA contract. SKETCHER was implemented entirely in Ada as a prototype version that could provide an early, if somewhat rudimentary, capability. Another software tool that was developed with IR&D funding for another project is the Ada Statement Analyzer (ASA). The ASA provides a statement count of all statements and clauses used in a compilation unit by the sections of /M18183/ and has been used to collect the Ada source statement statistics on this project.

Several other tools, had they been available, would have been of significant assistance during the design, code/debug and integration/test phases. These tools could have contributed significantly to improving software productivity by eliminating rather tedious and unrewarding tasks such as formatting the Ada source code to achieve pretty printing. A list of these tools is given in Section 5.

4.2 ADA LANGUAGE EVALUATION

This section addresses problems or inconsistencies with the use of Ada, including specific language factors and also more general Ada education issues.

4.2.1 Ada Language Factors

The following sections address features of the Ada language which seem to be problematic in one way or another. For example, the features may be difficult to understand or they may contain limitations in certain situations. The elements in question are organized according to the sections of the LRM.

1. Introduction - Problems exists in learning the Ada language, since LRM is difficult to read as a user document. A number of sections prove to be misleading in that the examples are either highly stylized, excerpted or not totally

AD-A152 314

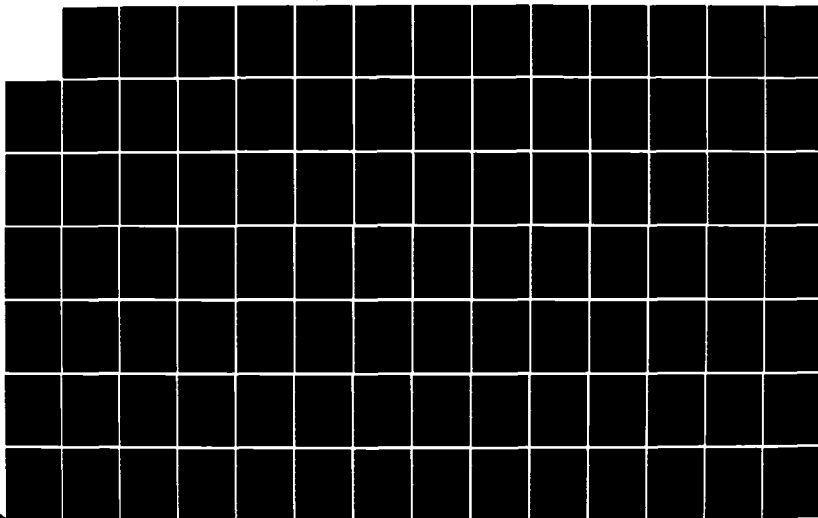
EVALUATION OF ADA (TRADEMARK) AS A COMMUNICATIONS
PROGRAMMING LANGUAGE VOLUME 1(U) SYSCON CORP SAN DIEGO
CA A L BRINTZENHOFF ET AL. 01 MAR 85 DCA100-83-C-0029

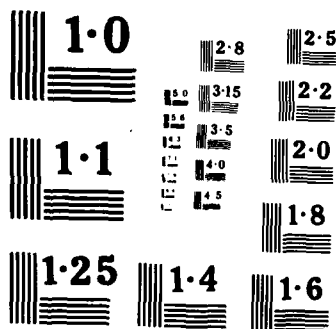
2/3

UNCLASSIFIED

F/G 9/2

NL





representative of the textual material. Information is either distributed and needs to be accumulated to understand the issue at hand or is available in weak form in that it needs to be deduced or inferred from the given material. Although textbooks are numerous now, examples tend to be tedious and may not adequately address the issue at hand. Some other means of quickly learning Ada and being able to locate basic information is required.

3. Declarations and Types - The declaration of variant records presents problems in certain instances. Components with the same identifiers cannot be multiply declared within the same record even within different variants of the records, since it is the record structure itself which establishes the scope and not the variant structures. Identical (syntactical and semantical) components which occur in different variants of a variant record therefore must be named differently. This has the potential of causing confusion and presenting a maintainability and reusability problem.

9. Tasks - The dichotomy between the specification of the entry points of a task in a task declaration which is visible to a task caller and the actual architecture of the accept statements of the corresponding task body, which generally will be hidden from the user, is a potential source of problems. There is no way to determine that guard conditions are implemented as a result of examining the task specification. Consequently, there is no information to determine whether unconditional, timed or conditional entry calls should be made. The result is a possible impact on the design and performance of the system. To the extent that some entries are serial and outside a selective wait statement or that selective waits contain multiple accepts for the same entry or that accept statements are nested, the caller will know less about the particular implementation and the impact on his design and overall system performance. In general, this situation needs to

be explored further to determine to what extent such implementation-dependent choices influence design considerations, and then whether the Ada language should be modified or whether task-entry specifications should be augmented with ANNA or whether some task-entry architectures should be proscribed via design guidelines.

11. Exceptions - A particular difficulty with exceptions is that they are associated with a package and are made visible via the corresponding package specification. Consequently, they are only weakly associated with their source or sources. Unless some type of additional association conventions are established, it is impossible to determine which visible entities can raise which exceptions. This can result in the needless proliferation of exception handlers in calling modules to cover all possible circumstances. More problematic is the situation where an exception may or may not be raised during a rendezvous for the same entry when that entry occurs in multiple accept statements. A similar argument applies to overloaded subprograms declared in the same package specification.

Appendix B-Pragmas - A pragma OVERLAY will be required to obviate recopying data to achieve change in representation (4x5 bytes vs. 2x10 bytes and contiguous) in a manner that is guaranteed to function correctly in all cases. The LRM specifically states that achieving overlays via the use of address clauses results in an erroneous program.

Table 4.2-1 indicates a qualitative assessment of how well the Ada language satisfies both general and specific requirements which are required by the applications.

Table 4.2-1. Software Application-Dependent Performance Requirements

GENERAL REQUIREMENTS		SPECIFIC REQUIREMENTS	
N	VERY HIGH PERFORMANCE	I/H	BIT/BYTE STRING ACCESS AND MANIPULATION
N	CAPABILITY TO INTERFACE WITH AND MANIPULATE SPECIALIZED HARDWARE	N	INSERTION OF ASSEMBLY LANGUAGE CODE
H, C, I	HIGH PORTABILITY OF SOURCE CODE	I	ACCESS TO OPERATING SYSTEM FUNCTIONS AND PRIMITIVES
H	SOPHISTICATED DATA STRUCTURES	I	ACCESS TO AND CONTROL OF INTERRUPTS
H	SOPHISTICATED CONTROL STRUCTURES	I	ACCESS TO REAL-TIME CLOCK AND ASSOCIATED TIMER(S)
C, I	VERY HIGH RELIABILITY	H	MACRO DEFINITION AND GENERATION*
		N	GENERATION OF I/O TABLES
		H	MODULARITY
		H	PARALLEL PROCESSING CONSTRUCTS
		H	STRONG DATA TYPING
		H	STRUCTURED PROGRAMMING CONSTRUCTS
		H	DATA AND CONTROL ENCAPSULATION*
		C	FORMAL VERIFICATION OF SOURCE CODE*

KEY
 I - LOW
 M - MEDIUM
 H - HIGH
 C - CAUTION
 REQUIRED
 I - IMPLEMENTATION
 D - DEPENDENT
 P - POTENTIAL EXISTS
 N - NOT EVALUATED

*—Specifically Trusted Software

4.2.2 Ada Education Factors

To use the features of the Ada language effectively, one needs to have a significant understanding of software engineering principles. Without such an understanding, the Ada language may result in producing programs which are Ada programs written in a FORTRAN, JOVIAL or CMS-2 dialect. There must exist a philosophy of identifying error conditions and responses to error conditions as integral parts of the design and the design process, not as afterthoughts added only when programs malfunction. The fact that Ada provides a facility to achieve this in terms of exceptions in no way assures that the facility will be used effectively.

It is necessary to have a methodology and software tools context for consistent application of the software engineering principles and the Ada language. Without this context, it is likely that Ada will be misused and misapplied. A methodology and supporting tools allow for significantly improved designs by providing constraints on required information, by consistently formatting the information and by permitting multiple, alternate views of the designs and the information implicit in those designs.

After the "basics" have been mastered, there is the need to deal with restrictions imposed on the full Ada language to achieve transportability, reusability, trustedness, machine verification and other application-dependent goals. On this project, significant effort and discussion were given to reviewing alternative communications protocol architectures regarding placement of the system management functions. How to partition and parameterize software to produce useful generic units will require considerable time and effort beyond the normal design. If transportability and reusability are not software quality criteria from the beginning of the project, they will not be achieved as by-products of any design or methodology.

4.3 SOFTWARE ARCHITECTURE ANALYSIS

This section presents the analysis of the software at the software system architecture level, and at the compilation-unit architecture level.

4.3.1 Software System Architecture

This section presents the inter- and intra-virtual package software architecture analysis for the communications protocols and trusted software application.

4.3.1.1 Communications Protocols Software Architecture Analysis

The Communications Protocols application was initially required to implement the Segment Interface Protocol (SIP), and the Mode VI subset of the Advanced Data Communications Control Procedures (ADCCP). The generic Open System Interconnection (OSI) Reference Model was utilized to structure the Communications Protocols system into a five-layer architecture and lower-level system management functions. Figures 4.3-1 and 4.3-2 summarize the processes that achieved the initial detailed architecture. The application was required for the transition to the Transmission Control Protocol (TCP), the Internet Protocol (IP) and the Asynchronous Balanced Mode subset of the Advanced Data Communications Control Procedures (ADCCP) later in the project.

A basic assumption in the requirements of the communications protocols system was that communication system software provides resource sharing types of services to user and end-process software. An assumption of scarce resources was made which implies both economy and management of system resources. The communications protocols system was to include environments that addressed distributed/multiple user components, host application components, and simulated interface component capabilities. The

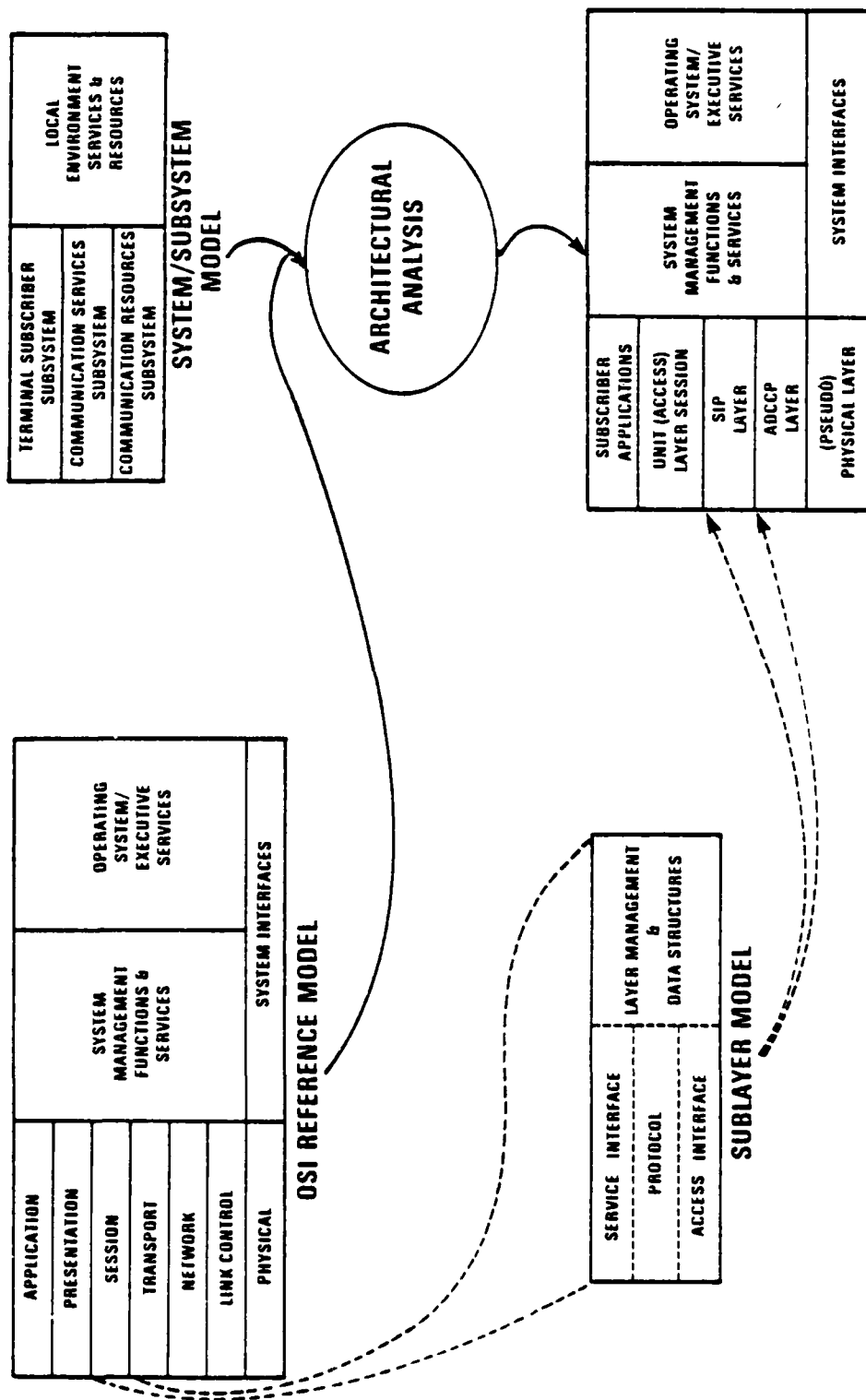


Figure 4.3-1. System Architecture Model Development

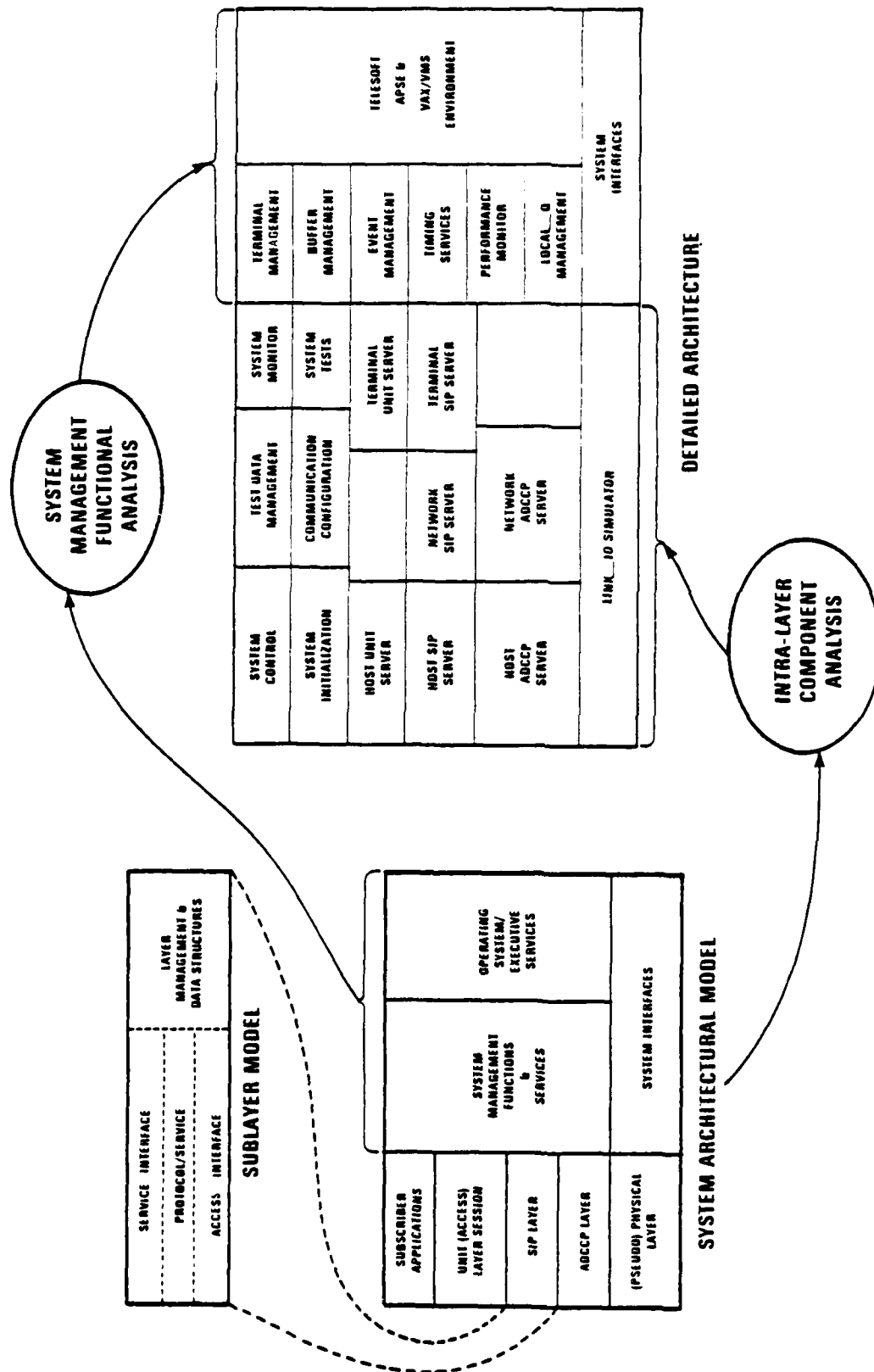


Figure 4.3-2. Detailed Architecture Development

software architecture analysis was conducted on two levels, the inter-virtual package level, and the intra-virtual package level, using the previously defined software quality factors of Section 3.

4.3.1.1.1 Inter-Virtual Package Architecture Analysis

This analysis focused on the number of modules, functional types of modules, and the inter-connectivity characteristics of the system. Figure 4.3-3 presents the virtual package design diagram that resulted after the TCP/IP transition. The diagram is a one-to-one representation of the detailed architecture.

The virtual packages are derived from four subsystems that are represented as the exploded portions of the Figure 4.3-3. The Terminal Subscriber subsystem modules provide the man-machine interface for invocation of major testing, monitoring, configuration, initialization and termination operations of the system. The Communication Services subsystem modules encompass the protocol layer services provided by the system (TCP/IP/ADCCP). The I/O Simulator subsystem module simulates communication device interface and transmission facilities. The System Management subsystem modules provide low level resource sharing/resource management services, data structures, and interfaces that are common to entities residing in multiple layers. They shield the protocol layers from details of the local operating system/hardware configuration environment. The arrows on the figure indicate major control flow direction. Data flow is in both directions.

The 19 virtual packages represent a faithful capture of the detailed architectural requirements in Ada features and terminology at a high level of the design. The Test Data Management virtual package was not implemented because of compiler implementation/run-time support problems with disc file I/O operations. The inter-virtual package dependencies generally

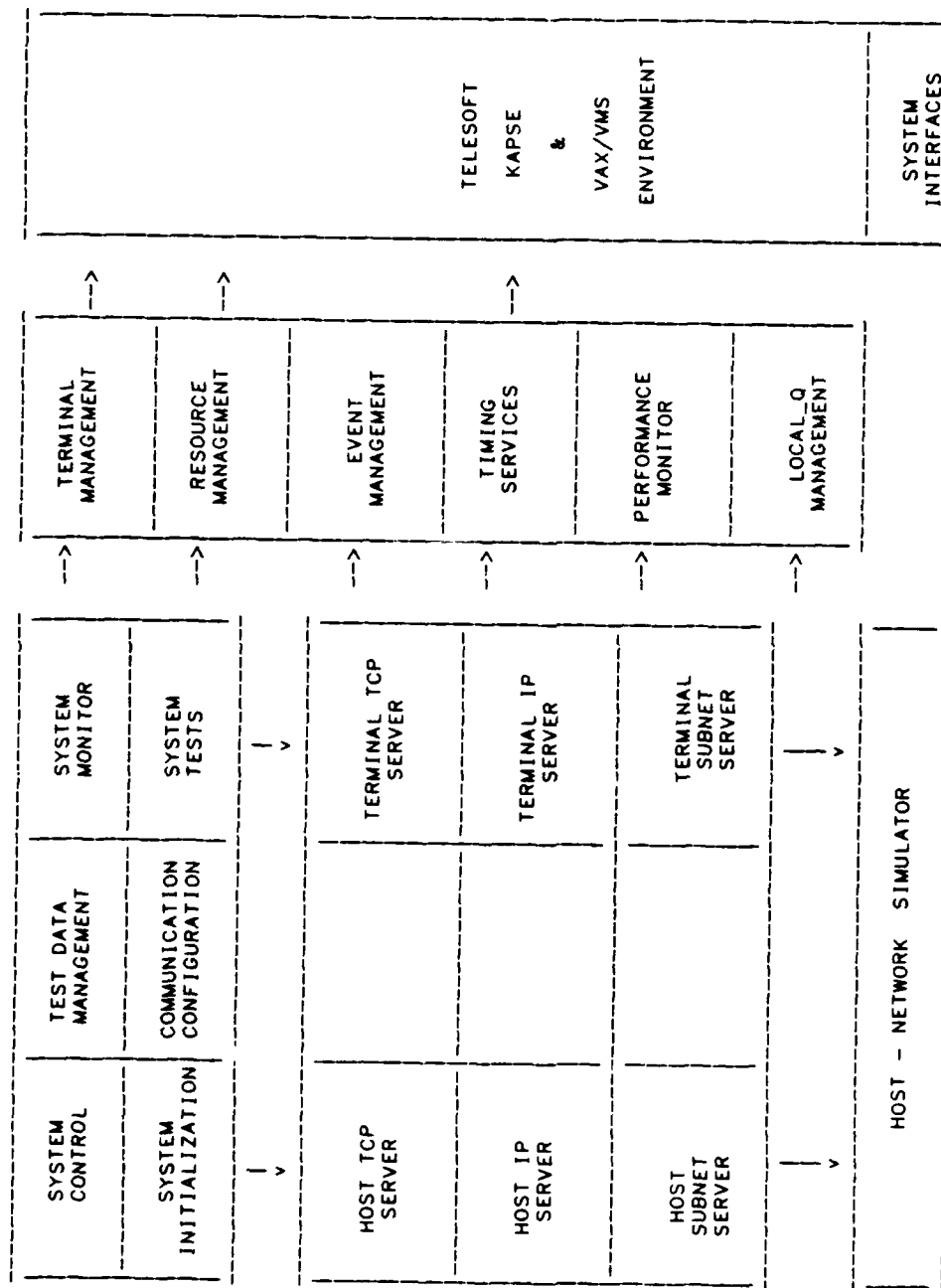


Figure 4.3-3. Communications Protocols System Detailed Architecture

follow the hierarchical organization of user/server relationships as inherited from the OSI model concepts. The calling pattern is strictly hierarchical, which was not the original intent of the system design. Implementing a more interrupt-driven design required the separate compilation feature of Ada which was not available with the compiler.

In the following inter-virtual package architectural analysis, each criterion is assigned a subjective value of excellent, very good, satisfactory, poor, or not evaluated.

CRITERION	EVALUATION
Communications Commonality	Excellent
Conciseness	Excellent
Consistency	Excellent
Data Commonality	Very Good
Generality	Excellent
Hardware Independence	Very Good
Instrumentation	Not Evaluated
Language Constructs	Excellent
Modularity	Excellent
Operating System Architecture	Not Evaluated
Operating System Independence	Very Good
Self Descriptiveness	Poor
Simplicity	Not Evaluated
Traceability	Excellent

The architecture exhibited excellent modularity characteristics which were evident in the transition to the TCP/IP protocols. Paragraph 4.1.9.1.2 of this section provides details concerning the transition.

4.3.1.1.2 Intra-Virtual Package Architectural Analysis

This analysis emphasizes the assessment of the virtual package modules in terms of high level Ada entities. The following presents the evaluation of appropriate criteria that apply to the software development quality factors cited in Section 4. The criteria are assigned subjective values of excellent, very good, satisfactory, poor, or not evaluated.

CRITERION	EVALUATION
Communications Commonality	Excellent
Conciseness	Satisfactory
Consistency	Excellent
Data Commonality	Excellent
Generality	Very Good
Hardware Independence	Very Good
Instrumentation	Not Evaluated
Language Constructs	Satisfactory
Modularity	Excellent
Operating System Architecture	Not Evaluated
Operating System Independence	Very Good
Self Descriptiveness	Poor
Simplicity	Poor
Traceability	Very Good

Figure 4.3-4 illustrates the organization of the Communication Services Subsystem modules. The Ada package features capture the characteristics of the sublayer model extremely well.

4.3.1.2 Trusted-Software Software Architecture Analysis

Figure 4.3-5 illustrates the system configuration of the original ACCAT GUARD configuration and Figure 4.3-6 illustrates the processes of the original ACCAT GUARD software. Since all of these were not germane to the analysis relating to the trusted processes, a subset was defined as shown in Figure 4.3-7. These components and the necessary support components were then organized into the virtual package architecture of Figure 4.3-8. Figure 4.3-9 illustrates the interprocess message flow, and Figure 4.3-10 illustrates the interprocess transaction flow. In both figures, the various processes, file managers and interprocess communication (ports) processes are mechanized as Ada tasks. Finally, Figure 4.3-11 illustrates the interaction of the Downgrade Trusted Process with KSOS and the Read-Write-Interprocess-Communication (RWIPC) software. In the modified configuration the interfaces to the high and low hosts are emulated via identical MMI software contained in the HSNS and LSNS virtual packages.

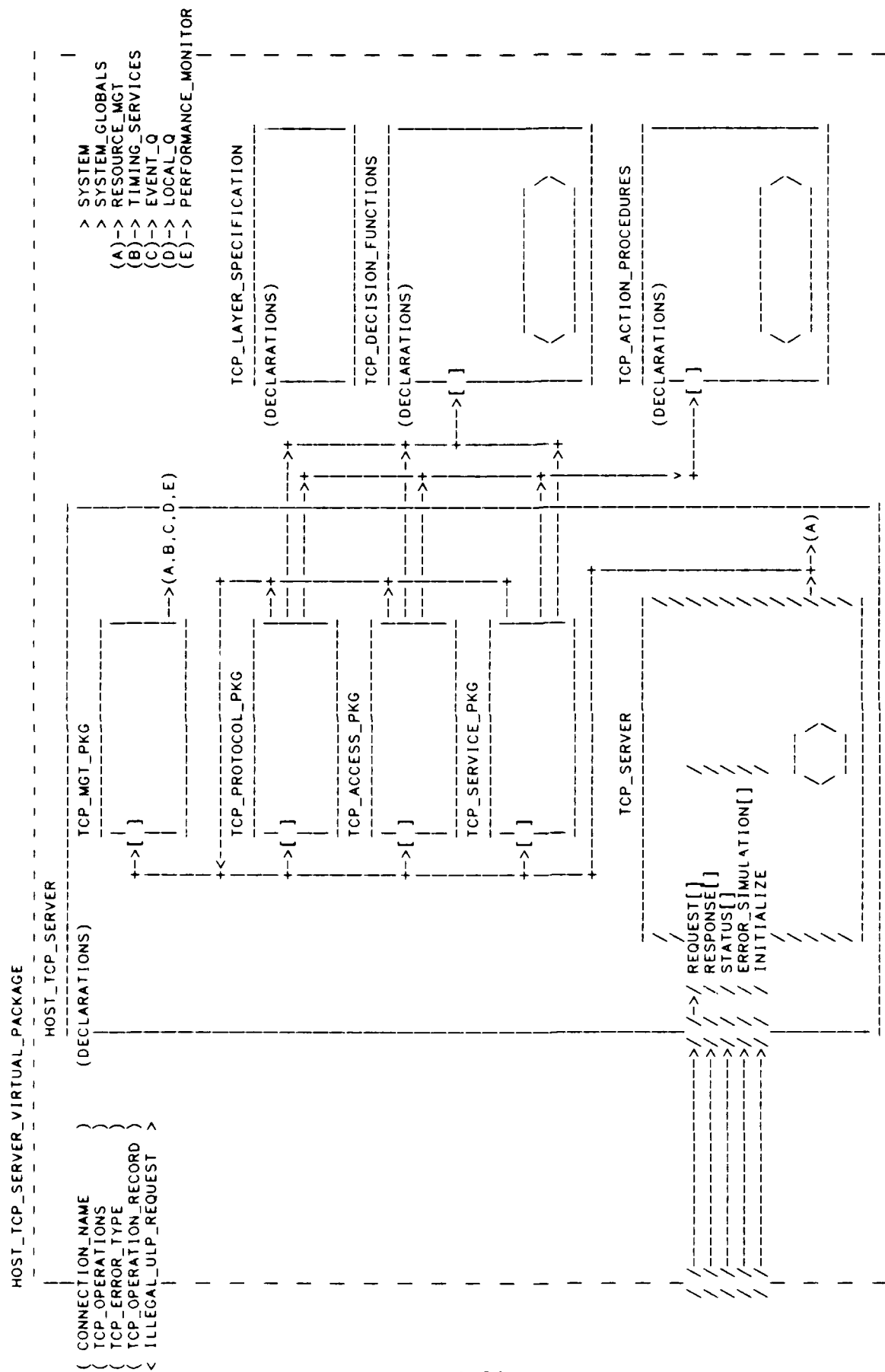


Figure 4.3-4. Host_TCP_Server Virtual Package Diagram

Table 4.3-3. Trusted Software System Software Statement Analysis Summary

Virtual Package Name	Number of CU's	Source Statements	Comment Statements	Total Lines
GUARD_GLOBAL_TYPE	1	54	99	170
GUARD_MASTER	1	19	17	34
HIGH_DOWNGRADE_DAEMON	1	140	91	251
HIGH_GUARD_SERVER_DAEMON	1	220	206	488
HIGH_SIDE_NETWORK_SIMULATOR	3	593	812	1790
LOW_GUARD_SERVER_DAEMON	1	120	111	278
LOW_SIDE_NETWORK_SIMULATOR	3	599	828	1807
TERMINAL_INTERFACE_SANITIZATION_PERSONNEL	3	924	809	2306
KERNELIZED_SECURED_OPERATING_SYSTEM	2	750	1106	2328
DOWNGRADE_TRUSTED_PROCESS	2	584	1603	2637
UPGRADE_TRUSTED_PROCESS	2	99	173	354
DATA_STRUCTURES	28	2673	3674	8862
TOTAL	48	6775	9529	21305

The following present the evaluation of the compilation unit analysis. Subjective values of excellent, very good, satisfactory, poor or not evaluated are assigned.

CRITERION	EVALUATION
Accuracy	Not Evaluated
Communications Commonality	Excellent
Communicativeness	Excellent
Completeness	Satisfactory
Conciseness	Very Good
Consistency	Excellent
Data Commonality	Excellent
Error Management	Excellent
Generality	Very Good
Hardware Architecture	Not Evaluated
Hardware Independence	Not Evaluated
Instrumentation	Excellent
Language Constructs	Very Good
Language Implementation	Not Evaluated
Modularity	Excellent
Operability	Excellent
Operating System Architecture	Not Evaluated
Operating System Independence	Not Evaluated
Self Descriptiveness	Very Good
Simplicity	Satisfactory
Traceability	Poor

Table 4.3-3 summarizes the compilation unit characteristics of each virtual package.

4.3.3 Compilation Unit Statement Characteristics

This section provides summary information on the number and types of Ada statements used across all compilation units of each application.

4.3.3.1 Communications Protocols System Compilation Unit Statement Characteristics

The statement characteristics of the compilation unit software are summarized in Table 4.3-4.

Table 4.3-2. Communications Protocols System Software Statement Analysis Summary

Virtual Package Name	Number of CU's	Source Statements	Comment Statements	Total Lines
SYSTEM_CONTROL	1	183	137	441
SYSTEM_INITIALIZATION	1	120	138	321
SYSTEM_MONITOR	1	326	260	810
COMMUNICATION_CONFIGURATION	1	228	246	638
SYSTEM_TESTS	2	714	632	1734
TEST_DATA_MANAGEMENT	0	0	0	0
HOST_TCP_SERVER	4	1814	2701	5772
TERMINAL_TCP_SERVER	1	379	483	1141
HOST_IP_SERVER	3	790	1909	3297
TERMINAL_IP_SERVER	1	341	452	1034
HOST_SUBNET_SERVER	3	1359	1442	3460
TERMINAL_SUBNET_SERVER	1	334	469	1028
TERMINAL_MANAGEMENT	1	481	324	1117
LINK_IO_SIMULATOR	1	145	114	325
PERFORMANCE_MONITOR	1	80	143	293
LOCAL_Q_MANAGEMENT	1	149	121	353
EVENT_MANAGEMENT	1	154	155	447
TIMING_SERVICES	1	248	182	549
RESOURCE_MANAGEMENT	2	286	401	914
TOTAL	27	8131	10309	23674

CRITERION	EVALUATION
Accuracy	Not Evaluated
Communications Commonality	Excellent
Communicativeness	Excellent
Completeness	Poor
Conciseness	Poor
Consistency	Excellent
Data Commonality	Excellent
Error Management	Poor
Generality	Very Good
Hardware Architecture	Not Evaluated
Hardware Independence	Very Good
Instrumentation	Poor
Language Constructs	Satisfactory
Language Implementation	Not Evaluated
Modularity	Excellent
Operability	Excellent
Operating System Architecture	Not Evaluated
Operating System Independence	Very Good
Self Descriptiveness	Poor
Simplicity	Poor
Traceability	Poor

Table 4.3-2 summarizes the compilation unit characteristics for each virtual package.

4.3.2.2 Trusted Software Compilation Unit Architecture

The final GUARD implementation does not faithfully reflect the intent of the original designs. Due to the compiler restrictions, modifications were made to the internal architecture of many modules to make them executable.

The virtual package DATA_STRUCTURES defined two "generic" packages to provide the ports, files, and locks for the high GUARD side and the low GUARD side. The ports, files, and locks were to be instantiated as entities in one of these two packages. Since generic instantiations are not implemented in the current compiler release, the two data structure packages were not created. The instantiations of the ports, files and locks were done independently via a pseudo-generic (text editor) approach.

values of excellent, very good, satisfactory, poor, or not evaluated is assigned to each of the criteria.

CRITERION	EVALUATION
Communications Commonality	Excellent
Conciseness	Very Good
Consistency	Excellent
Data Commonality	Excellent
Generality	Excellent
Hardware Independence	Not Evaluated
Instrumentation	Not Evaluated
Language Constructs	Excellent
Modularity	Excellent
Operating System Architecture	Not Evaluated
Operating System Independence	Excellent
Self Descriptiveness	Satisfactory
Simplicity	Excellent
Traceability	Excellent

4.3.2 Compilation Unit Architecture

The compilation unit software development statistics for the communications protocols and trusted software application are shown in Table 4.3-1.

Table 4.3-1. Composite Software Development Statistics

	Communications Protocols	Trusted Software
Virtual Packages	19	12
Library Units	26	25
Secondary Units	24	23
Statements	8,131	6,775
Comments	10,309	9,529
Lines	23,674	21,305

4.3.2.1 Communications Protocols System Compilation Unit Architecture Analysis

The following presents the evaluation of the compilation units on an individual criterion basis. Subjective values of excellent, very good, satisfactory, poor, or not evaluated.

compiler. A pseudo-generic approach was taken by using the text editor such that the "instantiations" of each generic package were created manually and compiled in a regular manner.

The criteria for the intervirtual package analysis are described below. Each criterion is assigned subjective values of excellent, very good, satisfactory, poor, and not evaluated.

CRITERION	EVALUATION
Communications Commonality	Excellent
Conciseness	Excellent
Consistency	Excellent
Data Commonality	Excellent
Generality	Excellent
Hardware Independence	Not Evaluated
Instrumentation	Not Evaluated
Language Constructs	Excellent
Modularity	Excellent
Operating System Architecture	Not Evaluated
Operating System Independence	Not Evaluated
Self Descriptiveness	Excellent
Simplicity	Excellent
Traceability	Excellent

The trusted and non-trusted processes are independent of each other. The only communication between them is through the port and file modules. The GLOBAL module is the package GUARD_GLOBAL_TYPES which contains data type declarations used throughout the system. No data objects were declared in this package. The G_MASTER is the GUARD_MASTER virtual package, representing the activation of the terminal drivers for the respective modules. The D_STRUCTURES is the DATA_STRUCTURES virtual package that consists of the ports, files, locks, the statistics module, and other utility modules.

4.3.1.2.2 Intra-Virtual Package Architecture Analysis

The intra-virtual package analysis focuses on the high level Ada entities associated with the virtual packages. The evaluation of the criteria for the software development quality factors at the intra-virtual package level are described below. Subjective

The Trusted Software system software architecture analysis addresses the architectural organization of the software modules from the inter-virtual package and the intra-virtual package perspectives. The characteristics of the virtual package architecture are addressed in terms of the criteria associated with the software development quality factors presented in Section 3.

4.3.1.2.1 Inter-Virtual Package Architecture Analysis

The major subdivisions of the system are the High Side processes, the Low Side processes and the Trusted processes. The High Side processes service transactions originating in the secure portions of the system (Secure Networks), sanitize these transactions and, if permissible, make them candidates for "downgrading" to the low side of the system (lower classification networks). The High Side processes consist of the Data Structures, High Downgrade Daemon, High Guard Server Daemon, Terminal Interface for Sanitization Personnel and High Network Emulator (MMI) virtual packages.

The Low Side processes service transactions originating in the low side of the system (lower classified networks) and transfers these transactions for servicing by the High Side processes. The Low Side processes consist of the Data Structures, Low Guard Server Daemon and High Network Emulator (MMI) virtual packages.

The Trusted Processes consist of the Downgrade Trusted Process, which transfers message from the high side to the low side, and the Upgrade Trusted Process, which transfers messages from the low side to the high side.

The functionality of several sets of the processes (the GUARD ports, files locks and the high/low user MMI operations) was identical with the result that these were excellent candidates for the Ada generic feature, which was not implemented by the

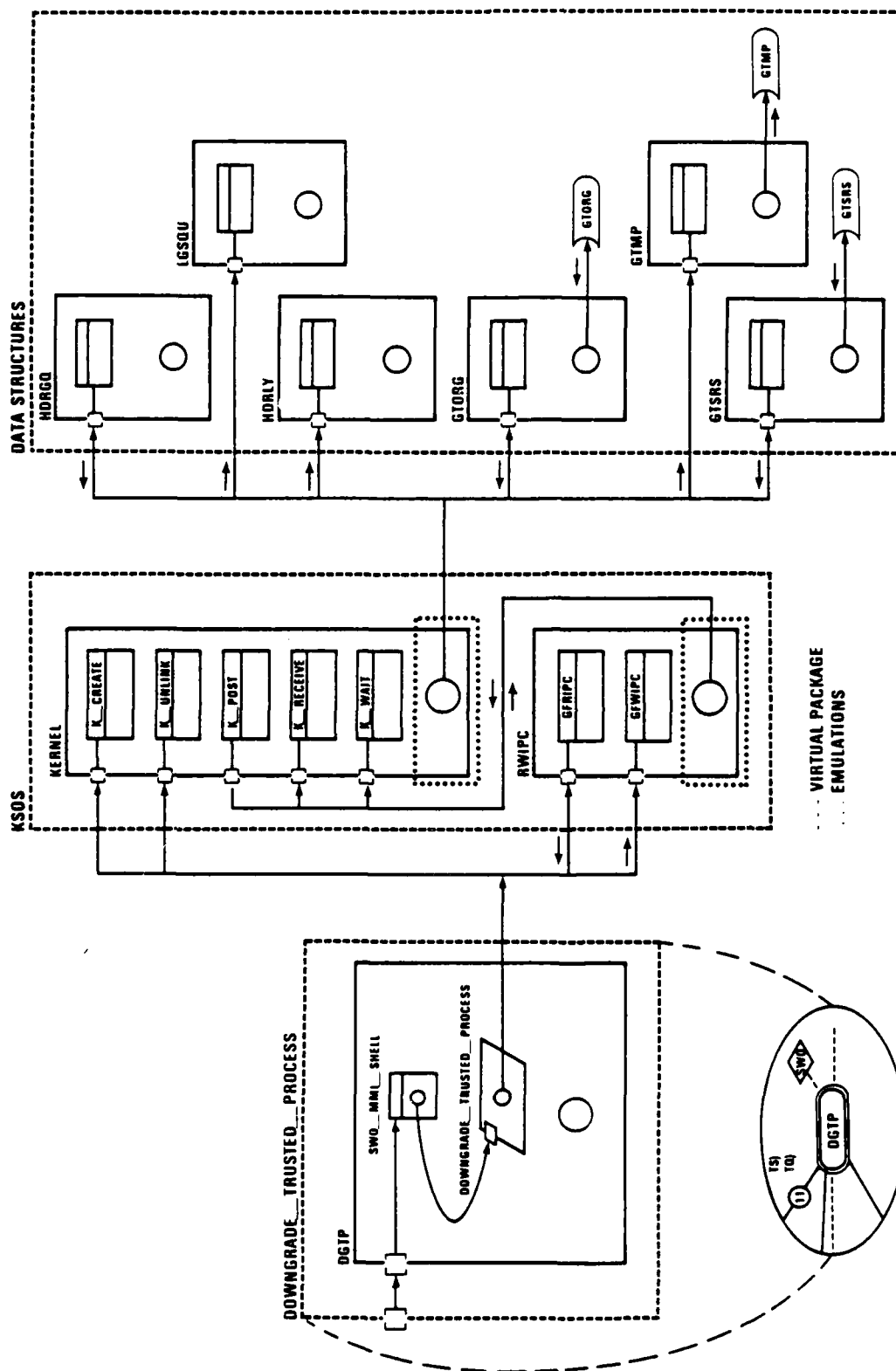


Figure 4.3-11. Downgrade Trusted Process Interactions

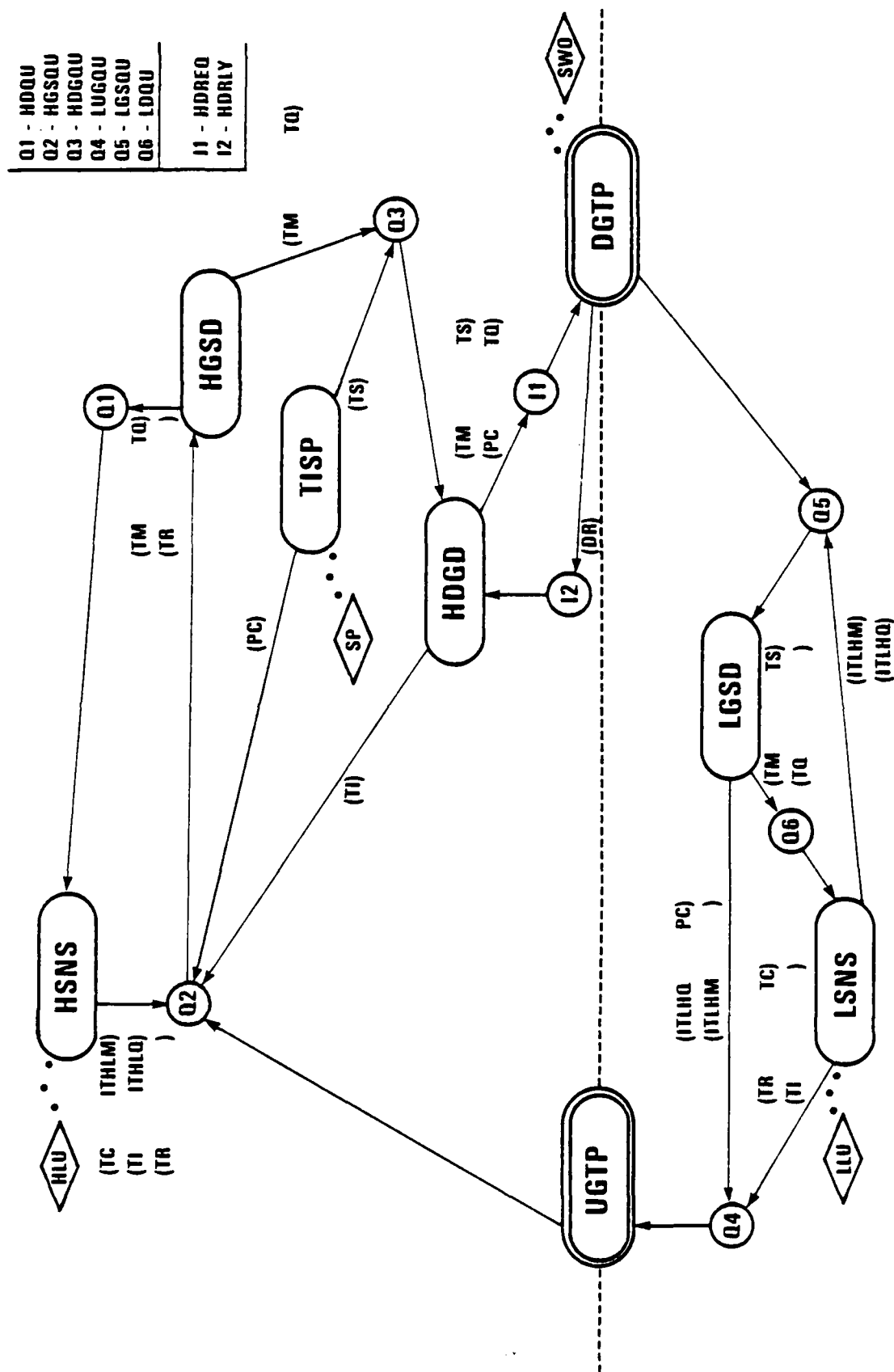


Figure 4.3-10. GUARD Transaction Flow

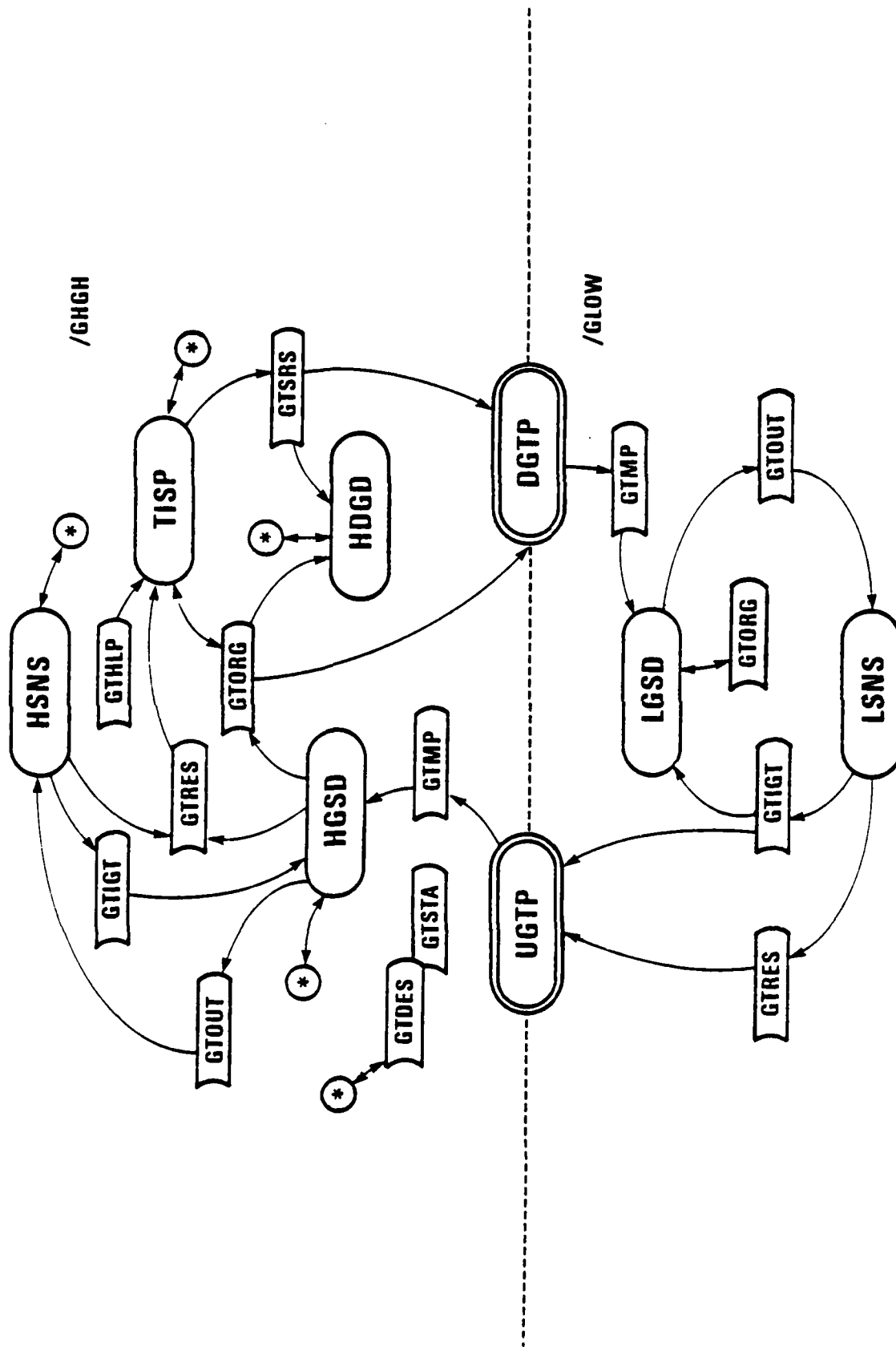


Figure 4.3-9. GUARD Message Flow

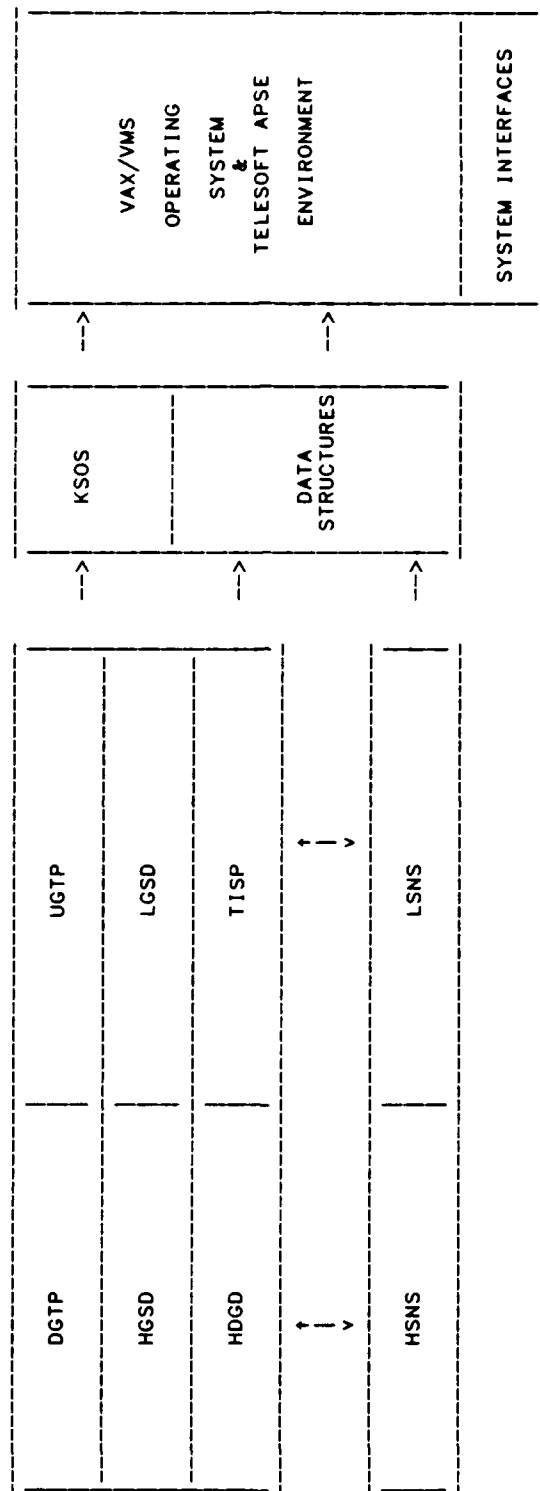


Figure 4.3-8. Trusted Software System Detailed Architecture

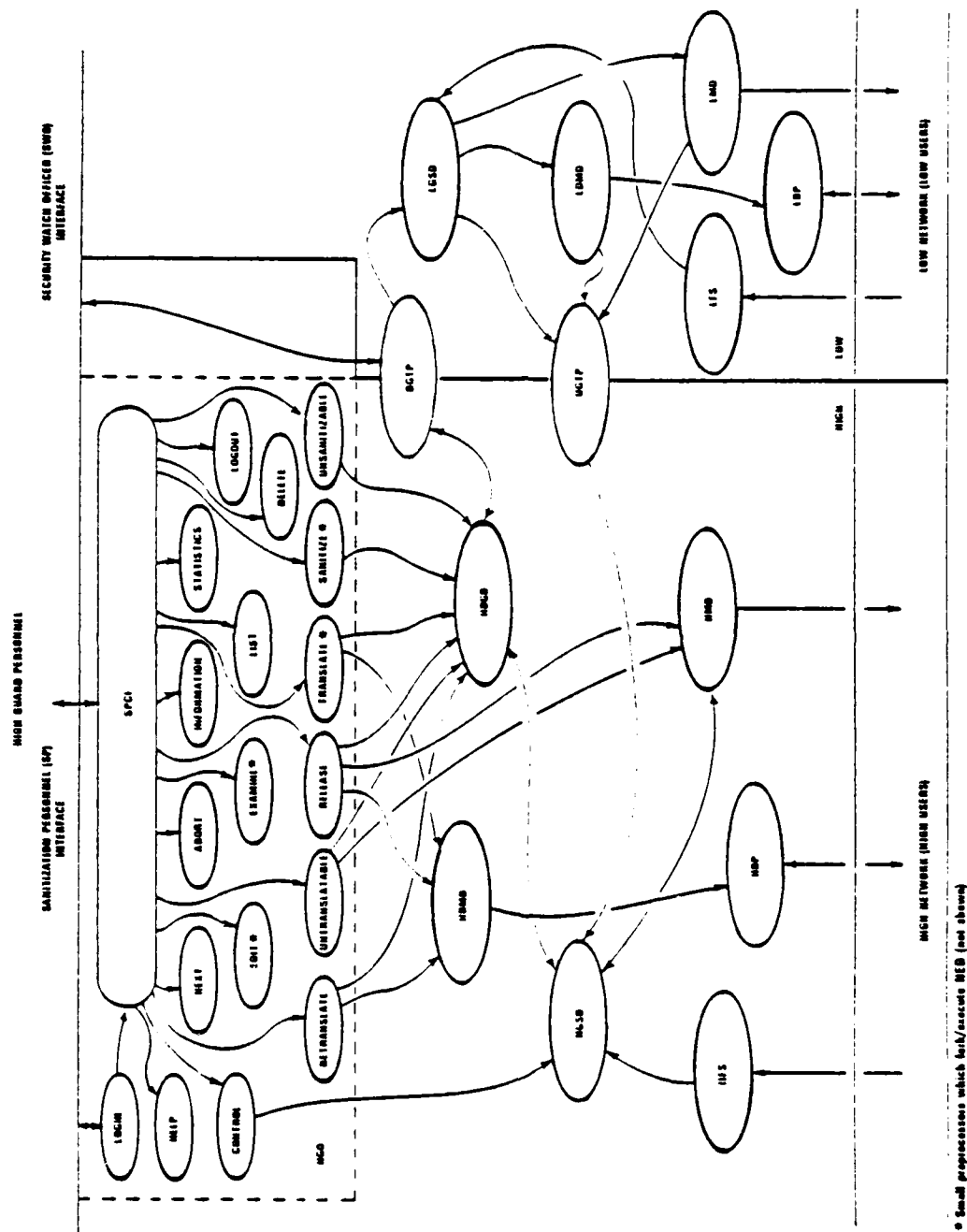


Figure 4.3-6. ACCAT GUARD Software

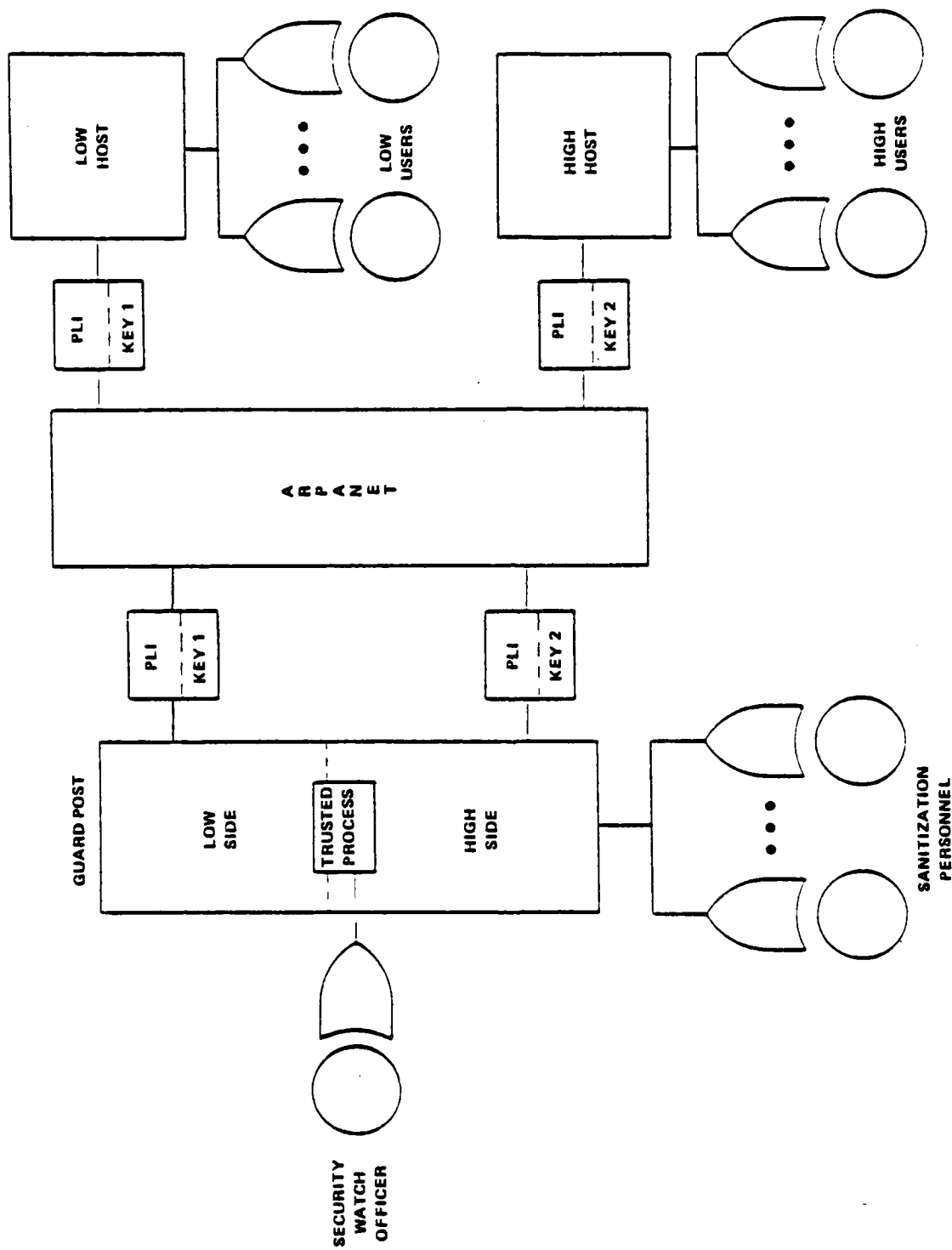


Figure 4.3-5. Original ACCAT GUARD System Configuration

Table 4.3-4. Communications Protocols System Aggregate Statement Statistics

LRM REFERENCE	Specifications	Bodies	LRM REFERENCE	Specifications	Bodies
2.0	COMMENT & PRAGMAS Comments (1) Pragmas (1)	2872 24	8.0	USE CLAUSE & Use Clause	54 5
3.0	OBJECT DECLARATIONS Object Declarations Number Declarations TYPE & SUBTYPE DECLARATIONS Type Declarations Subtype Declarations Derived Types Enumeration Types Integer Types Real Types Floating Point Fixed Point Array Types Record Types Access Types	7419 1 143 911 0 142 67 0 61 0 0 0 0 0 9 56 15	9.0	TASKS (***) Task Declarations Task Bodies Task Types (2) Task Objects Entry Calls (3) Accept Statements Delay Statements (2) Selective Waits Conditional Entry Calls (2) Timed Entry Calls (2) Abort Statements Terminate Statements Guard Conditions Rendezvous Block	75 13 11 28 0 11 0 86 0 22 0 0 0 1 81 77
4.0	ATTRIBUTES Attributes Allocators	109 6	10.0	WITH CLAUSES & SUBUNITS With Clauses Subunits (2)	111 0
5.0	STATEMENTS Assignment Statements If Statements Case Statements Loop Statements Block Statements Exit Statements Return Statements Goto Statements Labels SUBPROGRAMS Subprogram Declarations Subprogram Bodies Subprogram Calls(3)	1793 0 477 124 297 0 0 84 71 0 0 0 98 0 0 1516	11.0	EXCEPTIONS Exception Declarations Exception Handlers (4) Raise Statements	7 0 19
6.0	PACKAGES Package Specifications Package Bodies Private Types Limited Types	26 0 0 0	12.0	GENERIC Generic Declarations (2) Generic Instantiations (5)	0 0 59
7.0			13.0	REPRESENTATION CLAUSES Length Clauses (2) Enumeration Rep Clauses (2) Record Rep Clauses (2) Address Clauses (2)	0 0 0 0 0
STATEMENT COUNTS			1171 4759		
Number of Statements			6928		
Number of Lines			18915		

- (1) pragma SOURCE_INFO invokes a compiler specific error reporting facility
(2) Non-Implemented Ada feature
(3) Statistics tool cannot distinguish between task entry and subprogram calls
(4) Most exception handlers denote the error report facility (Reference Note 1)
(5) Generic instantiations reference UNCHECKED_CONVERSION and TEXT_IO Integer I/O

4.3.3.2 Trusted Software Compilation Unit Statement Characteristics

The statement size characteristics of the compilation unit software are summarized in Table 4.3-5. The smallest modules tended to be the support and utility packages. The largest was the Downgrade Trusted Process package (DGTP) since it was desired to preserve as much of the SPECIAL architecture as possible; otherwise, the DGTP could have been divided into smaller packages. The size of the ARPANET connection emulations (17%) consists mostly of the MMI drivers. The non-trusted processes, which were the High Downgrade Daemon (HDGD), the High Guard Server Daemon (HGSD), the Low Guard Server Daemon (LGSD) and the Terminal Interface for the Sanitization Personnel (TISP), were of a moderate size with the TISP being the largest. This was to be expected since their function is to sort and route the transactions from one place to another. The statistics for the number and types of statements used in the GUARD are in Table 4.3-5.

4.4 SOFTWARE PERFORMANCE ANALYSIS

This section addresses both general and application-specific software performance issues.

4.4.1 General Performance Characteristics

Very little was accomplished in assessing performance of the two applications. Considerable effort was devoted to struggling with compiler problems which complicated basic execution of both applications. Such problems included heap management problems involving task stack size and adverse interactions between DIRECT_IO operations performed from within tasks. Both resulted in program execution being terminated prematurely with few clues as to what was happening. Many features which could be used to assess performance were not available in the partial compiler

implementation. Such features included timed and conditional entry calls, lack of task priorities, and lack of pragma INLINE.

It was learned that the underlying tasking model for context switching was the run-until-block model. Once a task gained control it would retain control until some action, such as an I/O operation, caused the task to become blocked and force a context switch. This problem was circumvented by using a "null" task called intermittently from tasks to achieve a context switch. A second problem was detected with the activation of the multi-terminal capabilities. TEXT_IO was completely synchronous. An input request at a terminal resulted in all other activity in the system being suspended until that user entered the appropriate data. This is clearly unacceptable with regard to usability of the compiler and runtime support environment. This problem was never satisfactorily corrected. However, a partial solution was implemented for the sole purpose of permitting the development to progress so that as many features of both applications as possible could be implemented. The solution was to make slight modifications to the MMI and use the VAX type-ahead buffer capability to queue "null" commands which would permit the various tasks to receive control as context switches occurred.

It is important to note that although both of these run-time support environment aspects are specifically not addressed as requirements of /M18183/, they can directly affect the usability of an Ada compiler system.

None of the Ada-specific Efficiency-II criteria of Table 3.5-1, Section 3, were evaluated except for unchecked programming. Each criterion, depending on the nature and extent of its use, could significantly affect Efficiency-II.

4.4.2 Communications Protocols Performance Characteristics

Early in the system integration, several packages were reorganized as a result of a design review. This reduced the number of packages in the architecture and resulted in faster execution. Analysis determined that the program was being linked dynamically during execution and that execution was faster since there were fewer packages. This situation has some distinct negative implications with regard to how software architectures, which are designed for transportability, reusability and maintainability, may be influenced by performance optimization or run-time environment characteristics. At the very least, run-time considerations can not be ignored during the design process.

Unfortunately, because of compiler-related problems and limitations, little performance testing was accomplished on the communications protocols application. Little can be said about the software quality factors which affect performance. Tests which were planned are identified in Table 3.5-1, Section 3. There were several tasks and subprograms in which the pragma `INLINE` could have been used very effectively to achieve execution efficiency while at the same time preserving the overall software modularity. Significant, successful use was made of access variables and unchecked conversion for obtaining/managing buffer space from a compiler-supplied memory management package.

4.4.3 Trusted Software Performance Characteristics

All the tests indicated in Table 3.5-1, Section 3, were performed successfully. Stress testing, in terms of buffer/message saturation, and the emulation of KSOS-related errors were not performed. A compromising factor was that the original architecture was significantly altered to achieve an executing program given the compiler problems and limitations. The ability to preserve files across GUARD activations and assess the

recoverability aspects of this feature was not possible, since the files were implemented as memory-resident queues.

To evaluate the trusted software implementation with respect to Correctness, Integrity, Reliability, Robustness and formal verifiability, a validated, full-capability Ada compiler is needed. The original architecture can then be implemented and evaluated with respect to SPECIAL, the interpretation of the SPECIAL requirements, the suitability of the Ada features used, and the viability or necessity of placing further restrictions on the Ada language.

4.5 SOFTWARE ERROR ANALYSIS

4.5.1 Compilation Errors

A summary of the error types for both applications is given in Table 4.5-1 by generalized usage category. This information was gathered from early compilations and is a reasonable indication of what difficulties were encountered in the initial use of Ada. Typographical errors were not considered.

The majority of errors fall into four broad categories:

Undeclared Identifiers (25%); Improper Type, Subtype, Object Declarations (20%); Unresolved Subprogram, Task Entry Calls (15%); and Type Conflicts in Executable Statements (11%).

The first of these can be attributed primarily to carelessness in the need to declare all objects before they are used, failure to specify proper context and use clauses, and simple misnaming problems. The second can be attributed to the detailed syntax formats required by the type, subtype and object declarations where both constrained and unconstrained types are intermixed and the fact that unconstrained types are permitted in some instances, but not in others. The third can be attributed to the failure to provide context and use clauses in order to achieve the appropriate visibility. The fourth category indicates an

Table 4.5-1. Compilation-Related Errors

	NO. OR ERRORS	PERCENTAGE
*UNDECLARED IDENTIFIERS	67	25.0
*TYPE, SUBTYPE, OBJECT DECLARATIONS	54	20.0
*UNRESOLVED SUBPROGRAM CALLS, TASK ENTRY CALLS	40	15.0
*TYPE CONFLICT IN OBJECT ACCESSES	29	11.0
*IDENTIFIER NAMING INCONSISTENCY	10	3.8
*ERRORS IN USING ATTRIBUTES	7	2.6
*ILLEGAL USE OF RESERVED WORDS	7	2.6
*CASE STATEMENTS	5	2.0
*ARRAY INDEX ERRORS	5	2.0
*RETURN STATEMENTS	4	1.5
*INCORRECT/NON-EXISTENT TYPE-MARK WHEN USING QUALIFIED EXPRESSIONS	4	1.5
*FAILED TO IMPORT OR ESTABLISH DIRECT VISIBILITY	4	1.5
SUBTOTAL	236	89.0
OTHERS	29	11.0
TOTAL	265	100.0

initial lack of awareness of the implications of strong typing and the need to assure compatibility of the types at the object declaration level.

4.5.2 Execution Errors

The error data gathered during the execution of the programs, both during debugging/testing and software integration, were not as systematically collected as were the compilation data. The difficulties with the compiler would have required considerable effort to completely sift the compiler related errors from the programming errors. In addition, the desire to obtain executing programs resulted in placing emphasis on reorganizing the software architectures either at the system or module level in order to progress, and this would have presented another obstacle to error data collections and analysis.

However, some noticeable error patterns were detected. A number of errors were caused because default initialization values were not provided as directed by the programming guidelines. In the communications protocols application, a significant number of problems with exceptions were encountered. There was a failure to include exception handlers or to use the compiler-provided exception handling and reporting facility. In the trusted software application, exceptions were incorporated more systematically from the outset of the design, due to their being specified directly in the SPECIAL specifications. The exception reporting facility was used systematically as the only available debugging tool other than user-produced execution traces accomplished via TEXT_IO.

Several one-of-a-kind errors were encountered. During initial integration on the communications protocol application, elaboration errors were encountered due to: 1) incorrect context clauses being specified, 2) incorrect or out-of-date code files being detected, and 3) objects being initialized to out-of-range

values. Several standard Ada exceptions were also encountered:

- 1) TASKING-ERROR resulting from debugging activities,
- 2) CONSTRAINT-ERROR resulting from assigning out-of-range values or misuse of discriminants, and 3) Access checks resulting from reference to access objects with a null value.

Numerous run-time-system errors were encountered which were generally uninformative as to the cause. They were generally related to task stack overflow conditions which occurred as a result of a task containing too many nested subprograms, subprogram calls, too much inline code or interaction with DIRECT_IO. Once the cause was isolated, the software architecture was modified, if possible, to reduce the problem.

4.5.3 Software Error-Architecture Correlation

4.5.3.1 Communications Protocols

Some global variables were used to communicate overall system status between application layer entities and lower level tasks. Without the use of pragma SHARED on these variables, the program was clearly erroneous. Although this was thought to be the source of some early errors found during debugging, integration and testing, it turned out not to be the case since code optimization was not occurring. This problem did not occur in the trusted software application for two primary reasons. First, the use of global data was completely eliminated because of the nature of the software. Second, the interprocess communication was implemented via the sending and receiving system-control transaction which received the same routing and control as other transactions except that they could affect the state of the system depending on Security Watch Officer (SWO) actions.

Another area in which error types can be correlated directly with the architecture is in the case of exceptions. To the extent that the software architecture is not designed to handle

exceptions, the architecture will not be very reliable, modular or robust with regard to error processing.

Another type of error occurred in which tasks used as a resource monitor were accessed via one or more encapsulating subprograms with the data normally manipulated by the task contained in the enclosing package body. In some instances, the data were manipulated by the subprograms directly as opposed to by the encapsulated task, with the result that the data would not always be assured of being correct. The solution is to not use mixed-mode data accesses and to place all controlled data within the task itself if system limitations permit.

4.5.3.2 Trusted Software

No specific errors occurred in the trusted software application that could be traced directly to architectural considerations, which indicated a misunderstanding of Ada principles or semantics.

4.6 PROGRAMMING SUPPORT ENVIRONMENT

This section provides information on the compile-time and run-time environments which had an influence on the project and which may have an influence on future projects.

4.6.1 Compile-Time Environment

Because of the limitations of the NYU Ada/ED translator-interpreter, it was used only to verify the results of the development compiler regarding syntactical and semantical correctness of the code, to prototype ideas or achieve basic understanding of features. The version used was the validated version, 1.1.

A validated development compiler was not received in time to be used on the project. Three successive versions of the development compiler were used in both applications. All three versions were partial implementations in that: 1) MIL-STD-1815A was not completely implemented, 2) features were not always implemented as per MIL-STD-1815A, and 3) there were numerous errors or unreasonable limitations within the features which required alternative designs or implementations. A summary of significant deficiencies and their impact is provided in Appendix D. Although some of these features and their disposition are more significant than others, each one has had some impact on the implementation and on the overall schedules. Their impact on design was minimal since the design direction was to proceed as if a full, validated MIL-STD-1815A compiler was available for implementation. Some of the features which caused significant consequences are summarized below.

In both applications, there were substantial opportunities to use generics for the definition of queue managers and other entities. The use of generics here would have saved considerable coding and debugging time, and in the trusted software application could have eliminated approximately 5000 lines of Ada source code that were "instantiated" manually.

Only one compilation unit could be compiled at one time. Both a package specification and its corresponding body had to be compiled in the same compilation stream with no other entities. Another lacking feature was the separate compilation of subunits; bodies of code which grew larger than anticipated within a package body could not be stubbed out as subunits with separate bodies. As a result, package bodies tended to be large and required considerable time to rework.

Several features were lacking in the tasking area. These included task types, conditional and timed entry calls and task priorities. Task declarations were limited to one level of

nesting downward from the outermost unit. Although none of these was catastrophic, in the aggregate they represented a large nuisance factor.

4.6.2 Run-Time Environment

Several problems were encountered with the run-time environment which resulted in delays and workarounds. The most significant was the limitation on the task stack size. The result was that calls to tasks which contained nested subprograms or large amounts of in-line code frequently resulted in an ambiguous system error being produced and the program terminated. A similar situation occurred in making DIRECT_IO calls from within tasks. Other run-time environment problems were that tasking was mechanized using a run-until-block mechanization and that TEXT_IO was mechanized using synchronous instead of asynchronous input. The run-until-block mechanization was circumvented; no truly effective alternative was possible for the synchronous TEXT_I/O problem.

A major difficulty is that many features dependent on the run-time environment are not specifically identified in the LRM nor are they required to be provided in Appendix F, Implementation-Dependent Characteristics. To the extent that run-time environment parameters are not known, there may be significant problems with planned software architectures and such problems may become visible only during debugging or, worse yet, during system integration.

SECTION 5

CONCLUSIONS/RESULTS

5.1 SOFTWARE DEVELOPMENT METHODOLOGY

This section presents conclusions on the software development methodology which was formed and used.

5.1.1 Macroscopic Design Phase

The three components of the macroscopic design phase, the virtual package concept, the object oriented design diagrams, and the macroscopic PDL, have worked very well. They have accomplished the goal of achieving early Ada awareness in the designs while permitting late commitment to details. They have provided visibility into the system software architecture at a very high level and fully supported the software engineering principles. The approach is compatible with the DOD-STD-SDS design documentation and easily adapted to the documentation standard. A summary of the compatibility levels is shown in Figure 5.1-1. The methodology proved readily able to support the use of existing models and requirements such as the OSI Reference Model and sublayer models in the communications protocols application and the translation of both English language and formal SPECIAL specifications in the trusted software application.

One major problem area which will require further analysis to achieve a more solid methodology is whether or not the PDL should be strictly compilable. If substantial quantities of TBD_INTEGER type declarations and TBD_CONDITION objects are used to make the code compilable, the code can become rather difficult and tedious to read, thus diminishing the overall utility of the PDL. Regardless of the outcome of this issue, considerable emphasis should be placed on achieving package specifications which are correct, complete and consistent at the conclusion of the

DEVELOPMENT PHASE	
SYSTEM/SUBSYSTEM DEFINITION	— SOFTWARE REQUIREMENTS ANALYSIS
MACROSCOPIC DESIGN	— TOP-LEVEL DESIGN
MICROSCOPIC DESIGN	— DETAILED DESIGN
CODE/DEBUG	— CODING, UNIT TESTING
SYSTEM/SUBSYSTEM INTEGRATION	— CSC INTEGRATION
COMPONENTS HIERARCHY	
SYSTEM/SUBSYSTEM	— CSCI
VIRTUAL PACKAGE	— CSC (TOP LEVEL)
Ada PACKAGE, SUBPROGRAM	— CSC (LOW LEVEL), CSC UNITS
DESIGN INFORMATION	
MACROSCOPIC DESIGN:	— SOFTWARE TOP-LEVEL DESIGN DOCUMENT
VIRTUAL PACKAGE,	CSC (TOP LEVEL),
OBJECT ORIENTED DESIGN,	DATA, DATA FLOW, CONTROL FLOW
Ada PDL	
MICROSCOPIC DESIGN	— SOFTWARE DETAILED DESIGN DOCUMENT
Ada PDL,	CSC (LOW LEVEL), UNITS,
Ada	DATA, DATA FLOW, CONTROL FLOW

Figure 5.1-1. Methodology Compatibility with DOD-STD-SDS

The virtual package concept, which exhibits some of the characteristics of the Ada package, is the primary tool for capturing the OSI Reference model concepts and the system requirements which are presented in the detailed architecture of the system. The criteria that inhibit an excellent evaluation in all the development quality factor areas are hardware independence, operating system independence (including the implementation run-time support mechanisms), and the self-descriptiveness criteria. In communication systems, communication device specific interfaces cannot be ignored. Dependencies on operating system/executive characteristics must be localized to the highest degree possible. Where the inter-virtual package architecture analysis identified poor characteristics was in self-descriptiveness. This evaluation was based on the choice of names for package names, object/type declarations, and package entry point names (tasks/subprogram declarations). A considerable amount of renaming activity occurred at all levels of the project. A strong naming policy needs to be formulated, implemented, and enforced across all development phases to properly capture the readability potential of Ada PDL and code.

The architecture exhibits a high degree of module coupling between the TCP/IP/ADCCP server modules and the System Management modules which reduces transportability characteristics and increases service interface complexity at modules below the TCP layer. An alternative architecture that could eliminate or reduce this coupling would be to define the various System Management modules as generics to be instantiated in each layer. Although this approach would enhance transporability and reusability, it could adversely affect Efficiency II. Some experimentation would be required to draw any firm conclusions.

SOFTWARE QUALITY FACTORS	SOFTWARE QUALITY CRITERIA										SOFTWARE QUALITY CRITERIA									
	A	C	C	C	C	C	C	C	C	C	A	C	C	C	C	C	C	C	C	C
EFFICIENCY	I																			
FLEXIBILITY																				
INTEROPERABILITY																				
MAINTAINABILITY																				
REUSABILITY																				
TESTABILITY																				
TRANSPORTABILITY																				

Legend: E - Excellent V - Very Good S - Satisfactory P - Poor N - Not Evaluated

Figure 5.3-1. Communications Protocols System Inter-Virtual Package Analysis Summary

To use Ada effectively, a solid software engineering basis is required to assure that such Ada features as packages and generics are properly understood in the context of transportable, reusable and modular software. Without such an understanding, many of the Ada features will be used improperly or suboptimally resulting in many potential Ada benefits not being achieved. To have an effective combination of Ada with software engineering principles, a software development context consisting of a well-defined software development methodology, compatible software tools and a compatible Ada programming support environment must be provided.

There will be no shortcuts to learning the Ada language because it is complex. Moreover, the ultimate objective is not just learning the language, but rather learning to use the language effectively to achieve software engineering objectives. The one-week syntax and limited semantics training course can be eliminated from consideration. A much more substantial course in a broader framework needs to be implemented.

5.3 SOFTWARE ARCHITECTURE

This section presents conclusions on the designed and implemented software architectures and summarizes results of the previous architectural analysis.

5.3.1 Communications Protocols System

The following paragraphs present the Communications Protocols system software architectural analysis summary and conclusions. Subjective evaluation weights of excellent, very good, satisfactory, poor, or not evaluated, are used.

5.3.1.1 Inter-Virtual Package Analysis Summary

Figure 5.3-1 provides the summary of the inter-virtual package architectural analysis documented in Section 4 of this report.

model and package TEXT_IO. The tasking model was the run-until-block model which places a significant portion of the burden for achieving context switching directly on the programmer. Although this model may be acceptable in some types of applications, such as navigation and guidance control programs in missiles, it is certainly not a suitable algorithm for implementing communications protocols in multiple layers in a communications node. The Ada standard does not specify whether the mechanization of TEXT_IO should be synchronous or asynchronous. For single-user applications, either one will be acceptable. Since both applications were implemented for multiple-terminal users, a synchronous TEXT_IO mechanization in which ALL tasks within the entire system wait for a given user input is not acceptable from a usability standpoint. Although these two examples could possibly be dismissed as artifacts of a prototype, incomplete Ada compiler, they are representative of the type of problems that can ensue from the lack of specification of such entities.

5.2.2 Ada Language Education

One significant difficulty encountered is simply learning the Ada language. This is difficult for several reasons based on project experience. Ada includes many features which were previously available only in experimental or very special-purpose languages. Individuals need to learn not only the syntax and semantics of Ada, but also the concepts and ramifications of the features. Ada also demands attention to numerous details ranging from top-level logical and lexical architecture considerations down to which components should be private or limited private. This becomes more difficult when features interact with each other. The Ada reference manual as it stands today is complete, consistent and correct, but, unfortunately, it is not very usable from a practical user viewpoint because of the highly precise language used to describe all the interactions and subtleties. Thus, it seems that some alternative form of a manual is required to make the use and learning of the Ada language by programmers who are users, not implementors, easier.

In the area of software performance, particularly in Efficiency-II, little was accomplished regarding actual evaluation of the Ada features. There were a few instances in which Efficiency-II features were used and several instances in which some of those features could have been used had they been available. Unchecked conversion was used considerably in the communications protocols application. Other features which could have been used had they been available were the pragmas PRIORITY, INLINE, OPTIMIZE, SHARED and SUPPRESS, and timed and conditional entry calls. One negative factor was encountered in the use of recursive subprogram calls in that task stack overflow occurred due to a significant number of recursive calls made from within the task itself. Although this was subsequently corrected by converting the recursive mechanization to an iterative one, the problem warrants consideration. To the extent that large numbers of recursive subprograms are used, the calling sequences are highly data dependent and exception management has not been properly addressed. The program, although correct, may not be very reliable.

One area of considerable concern is the implementation-dependent features such as the pragma PRIORITY and the declaration of representation specifications for record types. To the extent that these features are implementation-dependent and a compiler implementor is permitted to NOT implement those features and still have its compiler validated, considerable difficulties may result in attempting to produce transportable and reusable code. If representation specifications are not provided, encoding and decoding of protocol packet and frame headers will need to be done explicitly either in Ada or via assembly language routines, in which case the choice may directly impact transportability.

A major concern is those features or alternatives which are not specified by the Ada standard and are not identified as being implementation dependent. Two significant areas in which difficulties were encountered are the mechanization of the tasking

TRUSTED COMPUTING BASE CONCEPT

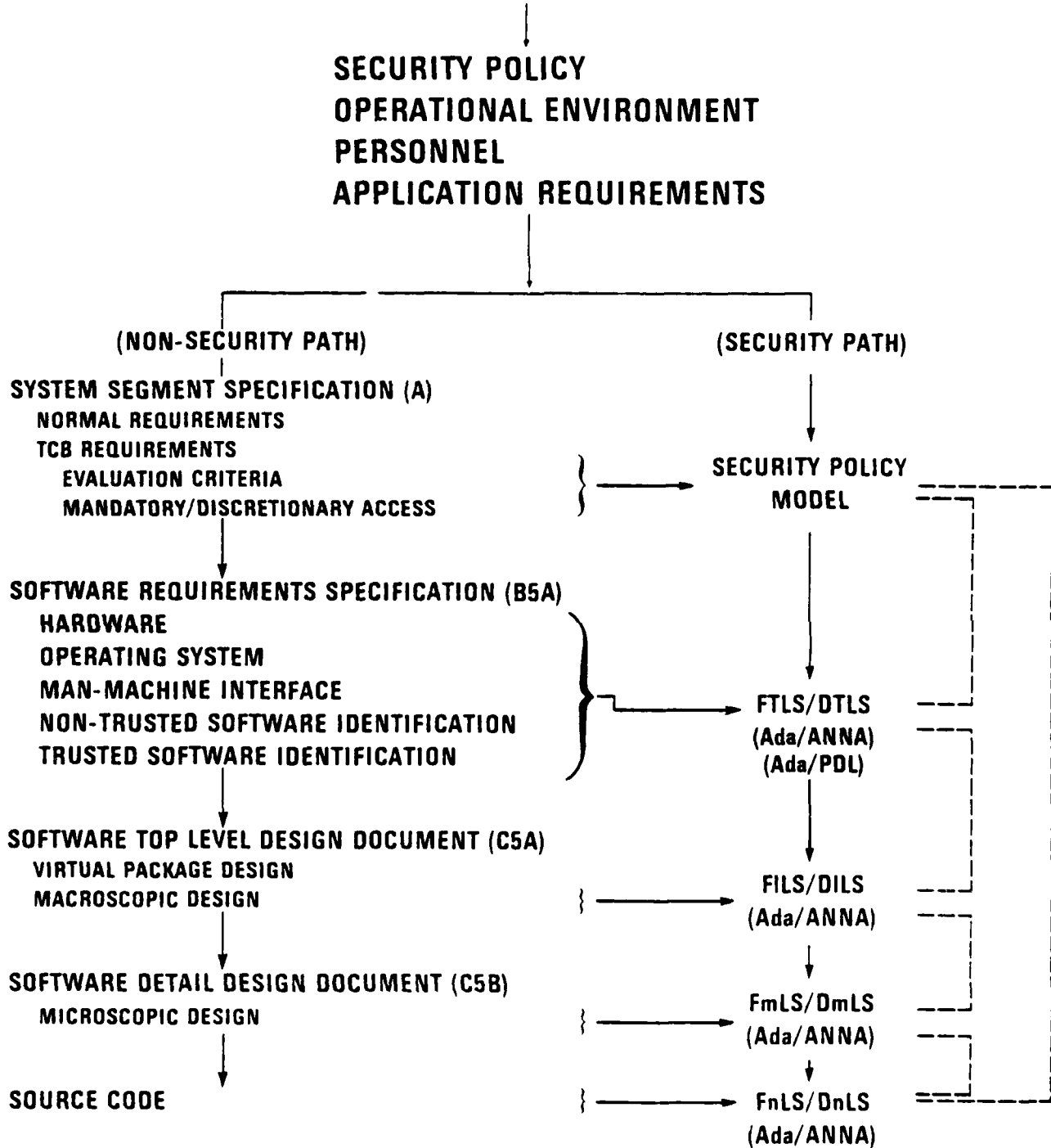


Figure 5.1-2. Trusted Software Design Methodology

implemented using access variables as a way of transferring the FROM_ULP and TO_ULP record structures. Since this mechanism provided both efficiency and flexibility in separating header and data portions of the segment.

5.1.8.2 Trusted Software

The design methodology functioned very well for the trusted software application with one exception. The primary difficulty was the translation of given SPECIAL requirements for the UGTP and DGTP into Ada effectively and at the correct level of detail. In terms of capturing the original requirements and translating them into Ada terms, the methodology was successful.

To produce a more effective methodology suited specifically to the development of trusted software, the methodology will have to be restructured into two parallel development paths, one for the trusted and the other for the non-trusted software. Separation must be established at a very high level, namely, the virtual package level, followed by further separation of visible and nonvisible portions of the trusted software at either the virtual package or object oriented design level. Figure 5.1-2 illustrates how such parallel development paths might be implemented and shows the successive refinements of the trusted software using Ada PDL and ANNA to produce the corresponding formal and descriptive top-level and n-level specifications.

5.2 ADA LANGUAGE EVALUATION

This section presents conclusions on the Ada language features which were used and on Ada language education and training issues.

5.2.1 Ada Language Syntax and Semantics

The features which Ada provides that can be used for producing top-level designs are excellent. These include such Ada-unique features as packages, generics, tasks and exceptions.

5.1.8 Application-Dependent Characteristics

This section presents conclusions on the design methodology for those cases in which the application characteristics have a specific influence on the methodology.

5.1.8.1 Communications Protocols

The design methodology was highly effective in permitting the OSI Reference Model concepts to be captured and refined through successive levels of detail into the final Ada code. The methodology is very capable in terms of taking generalized communications system requirements and translating them into effective software implementations which can achieve transportability and reusability. The graphical nature of the early portion of the design provides an excellent way of minimizing irrelevant details, but at the same time permitting key design decisions to be made highly visible.

The decision was made to reenter the TCP and IP specifications in their entirety since it was not clear which sections or subsections were to be used and to what extent. Another factor influencing this decision was that much helpful descriptive information was contained in the Specifications and that the Ada-like PDL had the strong potential of being assimilated directly in both the macroscopic and microscopic levels of design. Significant time was saved by reentering the Specifications in their entirety since this enabled portions to be used selectively, as appropriate, and in different phases of the design.

The second issue involving the degree of design information implied by the TCP and IP specification was resolved through subsequent discussion with DCA personnel from the Protocols Standards Group. They indicated that the intent of the specification was to assure functional compatibility of peer layers and not to necessarily achieve inter-layer interoperability within a single machine or to achieve overall transportability. Consequently, the interface was

design may then be well on the way by the time the full macroscopic design begins. Similarly, an overlap should occur between the macroscopic and microscopic phases, and between the microscopic and code/debug phases.

Based on the experience with the transition to the TCP and IP from the SIP and the modification of the ADCCP, the various levels of the design methodology work very well in supporting the identification, isolation and modification of units to meet changing requirements.

To provide more control of the level of detail in the macro and micro PDL, two factors need to be considered. The macro PDL needs to be reviewed carefully to be sure that it represents the correct level of detail, since both too much and too little detail will be harmful. Once this has been accomplished, and the level of detail has been determined to be correct, attempts should be made to use PDL expansion ratios to control the quantity of micro PDL. This will assure a reasonable progression from the macro PDL and provide a reasonable base to progress into the code/debug phase.

Another conclusion regarding the design methodology is that several small steps will be more effective in achieving a satisfactory design than a few large steps. It is possible to become prematurely involved in many Ada details that need not be considered to achieve the top-level design.

In conclusion, the macro/micro levels of abstraction, if they contain the correct level of detail, can assist considerably in making modifications to existing designs and code by enabling quicker identification of modules which are to be retained, deleted, modified and added. By following the macro/micro design for new requirements, it will be possible to retain the overall software architectures and follow the same methodology for the inclusion of the new requirements.

These should have been stated "Use [limited] nesting..." and "Avoid [excessive] package [and subprogram] nesting...". In many instances the guidelines will merely serve as guidelines and some judgement and interpretation will still be required. Additional guidelines need to be formed to specifically address transportability and reusability criteria and to identify such standard information items as programmer name, completion dates, and other data which may vary between facilities.

A preliminary set of trusted software design/programming guidelines has been formed. This set of guidelines is viewed as only the first step in a series to achieve a firm set of guidelines. The major problem is to form a set of guidelines that satisfy three fundamentally conflicting criteria: 1) being able to achieve formal verification of the designs and the implemented code, 2) being able to retain a useful subset of Ada constructs, and 3) being able to minimize unauthorized information flows via covert channels. To make further progress in this area, the full set of designs or a revised set based on the use of ANNA must be implemented. Extensive static and dynamic testing must also be conducted to determine the suitability of the restrictions and whether the set is complete.

5.1.7 General Software Development Methodology Considerations

The methodology has worked very well given the limitations under which the project was conducted. Several points need to be made regarding supplements to the methodology. A key component of the methodology is the need to overlap the phases in order to minimize interphase disconnects. Before completing the requirements phase, some amount of the macroscopic design should be initiated, probably at the virtual package and object oriented design diagram level, to provide an assessment of the completeness, correctness and consistency of the requirements as well as the feasibility of achieving a reasonable and satisfactory design. In this way, the requirements will receive a degree of validation, and some initial design prototyping will have been accomplished. Portions of the

concurrent processing situation with the expectation that most problems will be related specifically to tasking itself.

Whatever approach is taken, a sophisticated source level debugger will be required to effectively debug concurrent processing implementations. In order to have the debugging be efficient, access to such entities as task queues, task priorities, and task entry queues will be required so that experimentation can be conducted without the need for frequent recompilation.

Although the problems cited were aggravated by compiler problems, each application was self-contained and under the responsibility of a single individual. In large system applications which may involve hundreds of tasks distributed across several groups of programmers, a systematic way to achieve the integration will be required.

5.1.5 Design Guidelines

The design guidelines developed were generally effective. They were limited primarily to indicating what level of detail should be supplied at each level and did not explicitly address application-specific requirements. This area could be expanded further to address application-specific requirements with regard to performance issues, and software transportability and reusability issues. Additional design guidelines will also be required for implementing a distributed system where tasks may reside on different processors at different times.

5.1.6 Programming Guidelines

The programming guidelines developed were generally effective when used, but there existed some difficulty in interpretation. For example, two seemingly conflicting requirements were "Use nesting of modules in order to simplify or minimize overall compilation dependencies and number of compilation units." and "Minimize package nesting unless exceptional requirements exist."

were integrated and functioned in a serial manner, any subsequent problems resulting from the concurrent processing activation could be related directly to tasking activities. After the serial integration was accomplished, the transition to the full concurrent processing structure was accomplished with little difficulty except for the task stack size problems.

In the communications protocols application, the integration approach was rather different: the approach was to make an initial integration pass through all layers to establish basic interface communications, followed by another iteration to enable all the basic features of each layer, followed by another iteration to enable the special processing requirements (timeouts, missed/duplicate segments, etc.) and any remaining detailed requirements of each layer. A problem occurred in that as additional segments of code or tasks were activated, the characteristics of the software changed, with the result that new errors were encountered in areas which previously worked. Although some of these errors were traced to null access values and other programming errors, a significant number were also related to task stack size problems.

Based on these experiences which are heavily colored by the compiler problems, no firm conclusions can be drawn; however, some alternatives can be explored. A factor to consider is that each application includes approximately 30 tasks, and all tasks are continually active once the system has been completely activated. One approach to minimizing integration problems would be to have an executable PDL which allows task skeletons to be executed early in the design process. Another approach is to simply deactivate the code of the task bodies for the express purpose of achieving some initial integration. Another approach is to integrate and test small groups of tasks in their entirety and then integrate them into larger groups to the extent that this is possible within, for example, a virtual package boundary. Still another approach is to attempt to do the fundamental integration piecewise using serial processing to the extent feasible and then "convert" to the

5.1.3 Code/Debug

The coding process entailed no special difficulties in and of itself or with respect to making the transition from the microscopic designs, aside from the fact that some of the micro designs were underspecified.

Nevertheless, some other major difficulties had to be managed. The Ada compiler deficiencies, particularly at the detailed statement level, were detected incrementally resulting in initial confusion as to which problems were Ada problems and which problems were Ada compiler implementation problems. This was compounded by the fact that, initially, macroscopic- and microscopic-level PDL code was generated as if a full Ada compiler implementation were available, but the PDL was not actually compiled. There was also insufficient effort expended to conduct some rapid prototyping either for the purpose of becoming more familiar with the compiler, to resolve design issues, or to improve understanding of various Ada features.

The result was that early in the code/debug phase considerable time and effort were devoted to isolating Ada misunderstandings from compiler/run-time system problems and making the necessary corrections. Although separating Ada language problems from compiler/run-time support problems became less of a problem as coding progressed, other run-time support problems such as synchronous I/O and task stack size limitations surfaced that again caused confusion regarding the source of errors and methods of resolving them.

5.1.4 System Integration

Because of early difficulties with the compiler on the trusted software application, the software was integrated piecewise in a serial processing environment as opposed to directly attempting the integration and activation of the full concurrent processing architecture. In retrospect, it appears this approach was probably a wise choice. Once the various inter-virtual package entities

macroscopic design phase. A complete compilation of all library and secondary units should be required, even if some secondary units consist of null bodies, so that all entities and dependencies can be checked and the initial step can be made toward achieving system integration. The correctness, completeness and consistency of library units are particularly important. Otherwise, substantial refinement of the interfaces will be required during the microscopic design with the result that substantial recompilation of existing units will be required to resolve the deficiencies. Such recompilations will have a negative impact on schedules, costs, and the quantity of programming support environment resources required because recompilations may ripple through a large number of specifications until all problems are resolved.

Assessing the completeness of the PDL for any given entity is another problem to be resolved. The primary concern is to assure that top-level design, and only top-level design, is being accomplished. At this time, there is no specific indication as to how this problem should be managed, other than to require in-process design walkthroughs. An approach which may be viable is to iterate through the PDL two or three times covering all modules on each phase. Doing this would assure to some greater extent that each entity receives a more equitable proportion of the total design time and would diminish the possibility of some modules being designed in great detail while others are given only superficial treatment.

5.1.2 Microscopic Design Phase

The transition from the macroscopic design phase to the microscopic design phase progressed in a straightforward manner. One difficulty that was not detected at the micro design level was that, in several instances in the communications protocols application, considerable refinement was required at the coding level before the code could actually be written. Again, the issue is one of determining when the correct level of detail has been reached.

It is highly desirable for quality software to achieve information hiding to the highest extent possible. Communication system requirements generally must address the following system-wide considerations:

- o Security/Precedence/Priority Considerations
- o Performance/Resource Utilization and Management
- o Inter-layer Event Signalling/Scheduling

As such, the amount of globally visible data in the system is probably more than desirable. Information hiding was extremely successful at the compilation unit level where implementation details could be hidden from external user modules.

5.3.1.2 Intra-Virtual Package Analysis Summary

Figure 5.3-2 provides the summary of the intra-virtual package architectural analysis, documented in Section 4 of this report. There is a decline in Efficiency I primarily due to requirements for complex data structures and compiler feature limitations. Flexibility which is evaluated as very good, was exhibited when the TCP_Server modules became too large for the compiler to process and were reorganized. The subsequent reorganization of the TCP_Server virtual package architecture was effected quickly, accurately, and the overall modularity characteristics of the virtual package remained unchanged. Interoperability characteristics remain excellent. There is a significant falloff of the maintainability factor based on the evaluation of poor for the self-descriptiveness and simplicity criteria. The reusability and transportability evaluation of very good carries over from the inter-virtual package analysis summary. The testability factor falls to a value of satisfactory due to the poor evaluation of the self-descriptiveness and simplicity criteria.

5.3.1.3 Compilation Unit Analysis Summary

Figure 5.3-3 provides the summary of the compilation unit architectural analysis, documented in Section 4. The evaluation of the development factors remain the same as for the intra-virtual package analysis with the exception of the testability, which fell to an evaluation of poor. This resulted from the inability to insert instrumentation mechanisms such as TEXT_IO calls without significantly altering the performance/functionality characteristics of the software under test, and the inability to complete and use the Performance Monitor and System Monitor capabilities.

Because of the inability to complete several major functional areas of the communications protocols system, the evaluation of the software performance quality factors cannot be meaningfully accomplished.

5.3.1.4 Compilation Unit Statement Characteristics

Table 5.3-1 summarizes the compilation unit sizes of the developed software. Some generalities that can be made concerning the system statement characteristics are as follows:

COMPILATION UNIT SIZE	AVERAGE	LARGEST	SMALLEST
Code Statements	- 300	603	90
Comment Statements	- 381	1385 *	143
Total Lines	- 621	2480	293
* (Includes Remaining PDL)			

Communication Services	=> 60% of system code
System Management	=> 20% of system code
Terminal Subscriber	=> 19% of system code
Link IO Simulator	=> 1% of system code

SOFTWARE QUALITY CRITERIA	SOFTWARE QUALITY FACTORS															TRACEABILITY
	ACCURACY	COMMUNICATION	COMPLETENESS	CONSISTENCY	COMMONALITY	MANAGEABILITY	GENERABILITY	HITLER	ARCHITECTURE	INDEPENDENCE	INSTRUMENTATION	IMPLEMENTATION	MODULARITY	OPERABILITY	OPERATING SYSTEMS	
SOFTWARE QUALITY FACTORS	EFFICIENCY I			P								S				
	FLEXIBILITY					V				V			E			P
	INTEROPERABILITY												E			
	MAINTAINABILITY			P	E								E	N		P
	REUSABILITY						V			V			E		V	P
	TESTABILITY			P							P		E			P
	TRANSPORTABILITY									V			E		V	P
SOFTWARE QUALITY FACTORS	CORRECTNESS	N	P	E												P
	EFFICIENCY II							N				N		N		
	INTEGRITY	N		P										E		P
	RELIABILITY	N	E	E	E	P								E		P
	ROBUSTNESS				E	P	S									
	USABILITY		E													
														E		

Legend: E - Excellent V - Very Good S - Satisfactory P - Poor N - Not Evaluated

Figure 5.3-3. Communications Protocols System Compilation Unit Analysis Summary

Table 5.3-1. Software Development Statistics

	Communications Protocols	Trusted Software
Virtual Packages	19	13
Library Units	26	25
Secondary Units	24	23
Statements	8,131	6,775
Comments	10,309	9,529
Lines	23,674	21,305

Given an arbitrary guideline of 1,000 lines per module, 12 of the 27 compilation units were at or significantly over this guideline. The following are some observations concerning the overall module size characteristics of the system:

- o Large number of comments exist due to capture of TCP/IP specifications (This may or may not be desirable.)
- o Lack of generics/separate compilation features resulted in generally larger modules
- o Better performance with fewer packages resulted in generally larger packages
- o Reduction of recompilation dependencies influences overall compilation unit size
- o Workarounds at the code/debug phases tended to increase module size.

The modules over 1,000 lines could have been significantly reduced in size with the generic/separate compilation Ada features. It seems reasonable that library units and subunits should fall into or below the 1,000-2,000 line range consistently, especially when the separate compilation and generic features of Ada are used.

5.3.1.5 Other Observations

The analysis of the communications protocols system evaluated software quality criteria from three different perspectives: the inter-virtual package level (A1), the intra-virtual package level (A2), and the compilation unit level (C). It is of some interest to note the following:

CRITERIA	A1	A2	C
Communication Commonality	E	E	E
Conciseness	E	S	P
Data Commonality	V	E	E
Generality	E	V	V
Hardware Independence	V	V	V
Language Constructs	E	S	S
Modularity	E	E	E
Operating System Independence	V	V	V
Self Descriptiveness	P	P	P
Simplicity	N	P	P
Traceability	E	V	P

The communication commonality, consistency, and modularity criteria evaluation remained excellent through all three levels of analysis. This is due in part to the following:

- o Commitment to OSI Reference model and sublayer modeling concepts.
- o High degree of conceptual compatibility between protocol specifications and generic layered architecture concepts.
- o Ability to capture architectural concepts at a high level of design using Ada features.

The implementation was able to localize the hardware and operating system (run-time support) dependencies to a high degree. These criteria were consistently evaluated as very good. The self-descriptiveness and simplicity criteria were consistently evaluated as poor across all analysis levels. This is attributed to the poor choices for names, complex data structures, asynchronous control

flows, and implementation decisions based on Efficiency-II performance factors.

The deterioration of the traceability and conciseness criteria from excellent at the inter-virtual package analysis to poor at the compilation unit analysis is due to the following:

- o The initial macroscopic and microscopic design PDL was not compilable.
- o Failure to utilize the externally generated TCP/IP PDL statements correctly and at the correct level of detail.
- o The nature of the run-time system support of tasking (context switching) and I/O processes was not understood soon enough in the project.
- o The workaround during the code/debug and integration/test phases of the project could not feasibly be reflected back to the previous design level PDL.

This situation was a major factor in the recommendation for inclusion of compilable (and possibly executable) PDL at both the macroscopic and microscopic levels of design in the methodology.

One criterion, data commonality, was evaluated higher at the intra-virtual package and compilation analysis levels, than at the inter-virtual package level. In hindsight, one can conclude there was some difficulty dealing with the very specific and complex data structure declarations provided in the protocol specification documents at the earlier stages of the virtual package (macroscopic) level of design.

A major issue is the inclusion of significant amounts of TCP/IP specification text as comments in the code. This is extremely useful during development/maintenance activities. However, significant overheads are produced such as increased recompilation

time, larger listings to assimilate, and tracking the comments in source code with specification changes, deviations, and/or performance alternatives. There are clearly tradeoffs to be made.

A major issue arose concerning declaration of tasks in the specification of library units, namely that tasks not be visible in package specifications. The general consensus was that a task was an implementation decision and not a design decision. From the perspective of communication system design and development, points of asynchronous/event driven processing are an integral part of the design of the system, and not merely an implementation option. Additional issues associated with "hiding" tasks are that: 1) conditional or timed entry calls are not directly possible from user modules, and 2) user modules may needlessly declare tasks to monitor major data structures which are already guarded by a non-visible service module task. Finally, if tasks are visible, it will be easier to encapsulate them if desired then it will be to make them visible if they are initially encapsulated.

A major issue concerning the transportability characteristics of the system was debated throughout the life of the project. The issue centers on what "boundaries" are pertinent when the topic of transportability is being discussed. If the entire system is to be rehosted on a different host environment, the system management subsystem modules (package bodies) would have to be adjusted/modified to conform to the new operating system and/or run-time support interfaces and features presented while the remainder of the system could be transported directly. Since the system management subsystem comprises approximately 20% of the system, this rehosting could be considered transportable. If a layer of the system, say a TCP_Server module, was migrated and integrated to an existing system, along with the system management modules, then the system management modules would have to adjust as above; however, the percentage of code to change would increase to 50% or better (i.e., moderately transportable). If the TCP_Server module were migrated and integrated by itself (dependent system

management modules not provided), at a minimum the service sublayer, the access sublayer, and the management sublayer would have to adjust to the new environment. The OSI Reference model architecture together with the sublayer model, which was captured in the design and preserved in the compilation unit architecture, identifies and maintains these "transportability boundaries" to a high degree. The transportability characteristics of software can only reasonably be discussed relative to such boundaries.

5.3.2 Trusted Software System

The following paragraphs present the Trusted Software system software architectural analysis summary and conclusions. The summaries are presented according to the ordering of the analysis in Section 4. The summary assigns subjective evaluation weights of excellent, very good, satisfactory, poor, or not evaluated,

5.3.2.1 Inter-Module Architectural Analysis Summary

Figure 5.3-4 provides the summary of the inter-virtual package architectural analysis, documented in Section 4 of this report. The design and definition of the transaction data were ideal for the variant record structure. The flexibility of a single record type for use in all cases of transaction type formats increased the effectiveness of the system software architecture. Emulated generics were used whenever common program structures were identified. The instantiation of the ports, files and other structures allowed a single design to be used in several roles.

Exception handling was very successful at trapping error conditions. Errors were propagated to the calling entity as a raised exception condition, which proved to be simple, and processing control was greatly enhanced. The readability of the code was also increased since the exception handlers were explicit in their representations of error management.

SOFTWARE QUALITY FACTORS	SOFTWARE QUALITY CRITERIA										SOFTWARE QUALITY FACTORS									
	ACCURACY	COMMUNICATIONS	COMPLETENESS	CONCISENESS	CONSISTENCY	DATA	ALTERNATE	MANAGEMENT	GENERALITY	ARCHITECTURE	INDEPENDENCE	INSTRUMENTATION	LANGUAGE	CONSTRUCT	IMPLEMENTATION	MODULARITY	OPERABILITY	OPERATING	OPERATING	OPERATING
EFFICIENCY				E																
FLEXIBILITY									E											
INTEROPERABILITY																				
MAINTAINABILITY				E	E	E														
REUSABILITY									E											
TESTABILITY				E																
TRANSPORTABILITY																				

Legend: E - Excellent V - Very Good S - Satisfactory P - Poor N - Not Evaluated

Figure 5.3-4. Trusted Software Inter-Virtual Package Analysis Summary

The renaming capabilities of Ada allowed the references of the packages defined in the DATA_STRUCTURES virtual package to conform to the references defined in the requirements documentation, resulting in increased readability and traceability to design specification.

The architecture of the packages represents a clean, logical and straightforward approach. In general, the non-trusted modules are well organized and structured with respect to the number of packages and their specifications and with respect to the original requirements. The trusted processes, specifically the Downgrade Trusted Process, could have been structured more reasonably. Its architecture, however, was a direct result of attempting to preserve the architecture of the SPECIAL PDL and to address the issues of correctness and traceability.

The virtual package concept is the primary tool for capturing the trusted as well as the non-trusted process specifications along with the system requirements presented in the detailed architecture of the system. In almost all cases, the virtual package diagrams presented the features necessary for the trusted software clearly and correctly. Such features as separation of trusted and non-trusted processes, distinction of the IPC as the communication medium between the processes, and the ability to emulate the ARPANET connections and the KSOS interfaces proved to be accurate and descriptive. Imported and exported entities are well described.

5.3.2.2 Intra-Virtual Package Architectural Analysis Summary

Figure 5.3-5 provides the summary of the intra-virtual package architectural analysis, documented in Section 4 of this report. The values given in the intra-module evaluation fell slightly from the inter-module values. This is in part because of the run-until-block algorithm used by the run-time system. The specifications defined by the SPECIAL PDL did not provide the measure of

modularity that was deemed desirable by our guidelines. The Downgrade Trusted Process package (DGTP) was too large with respect to modularity, testability and maintainability. The choice made at this level was to be as close to the specifications of SPECIAL as possible; otherwise, this module could have been divided into approximately three separate packages.

5.3.2.3 Compilation Unit Architecture Analysis Summary

Figure 5.3-6 provides the summary of the compilation unit architectural analysis, documented in Section 4. The evaluation of the development factors for the compilation unit evaluation fell in value from the intra-virtual package analysis. The primary reason is a restriction of the run-time system whose implications were not realized at first. Each task has associated with it a stack/heap space which contains the task code and run-time parameters and the formal parameters and declared objects of all subprograms called by the task, directly or indirectly. An overflow condition occurred at almost every step of the debug/integration phase of the study. Recursion, which was used in the SPECIAL PDL, compounded the situation. Because of this restriction, the architectures of the compilation units were virtually destroyed in terms of modularity, information hiding and design guidelines. Another restriction is the case of synchronous I/O. Output to the terminal users by the individual terminal drivers was done on a first-come, first-served basis. This was to be expected and produced no problems. The input requests, however, would block the entire system until the input request from a specific terminal was supplied.

Because of these restrictions, the final Ada code is not totally traceable to the intended designs. The system does perform correctly, accurately and is stable. However, it is believed that with the changes that occurred, the principles of trusted software may have been compromised. Moreover, there is a little point, given the restructuring, in attempting to assess compromises that may exist since the architecture and total functionality are not

design. To resolve this issue and also address the performance aspects of the architecture, it is recommended that the full designs be implemented and evaluated and that alternative architectures be explored.

6.3.2 Trusted Software

The trusted software conforms to the original requirements except that the mechanization is multitasking as opposed to the original multiprocessing. At the intra-package level of design, many individual aspects of the original design had to be modified due to either compiler limitations or run-time support inadequacies. As a result, many tests whose results might affect the originally planned architecture were not conducted.

It is recommended that the intra-package architectures be restored to their original designs and the corresponding code be implemented. It is recommended that careful assessments be made of the multitasking vs. multiprocessing configuration and the nature and extent of interactions between trusted and non-trusted processes. This should be accomplished via extensive stress testing and the injection of errors into the UGTP and DGTP via the KSOS interfaces. Only through this approach can all levels of the architecture be assessed with regard to both specific trusted software criteria and the more general software quality factors of Maintainability, Testability, Correctness, Integrity, Reliability and Robustness.

6.4 SOFTWARE PERFORMANCE

Because of compiler problems, little opportunity existed for exploring alternative architectures and what their impact would be on performance. The impact of encapsulated vs. visible tasks on performance, and of a large number of task entry points with single parameters vs. a small number of entry points with multiple parameters were not explored, nor were the adequacy and performance

6.3.1 Communications Protocols

It is highly recommended that generic models, such as the OSI model, be utilized in a fashion similar to this project. Such architectural structures provide a relevant architectural basis that is expressible in Ada language entities and provide concepts that can be effectively applied at multiple levels of the methodology recommended by this report. Based on the use of these concepts, the Ada language provides a natural, and heretofore unavailable, design bridge between communication system architectures and requirements and the implementation of the system.

The TCP and IP Specifications, as well as future standardized specifications, should be made available on-line so that: 1) design and development efforts can be reduced by obviating the need for recreating the information, selecting a suitable format and verifying the information; 2) the Specifications can be made widely available at minimal cost; and 3) updates to the material can be quickly supplied. It is also recommended that anomalies in the Ada PDL be resolved. It is recommended that future specifications address the issues of transportability and interoperability more explicitly by indicating concisely if they are or are not, or to what degree they are to be met and precisely what constraints, if any, apply. In the case of the TCP and IP Specifications, future revisions of the document should assure that somewhat stronger and more precise statements are made to clarify the intent of the Specifications with regard to requirements and design options.

Because of compiler limitations, only limited information was obtained with respect to the transportability, reusability and performance characteristics of the software. One architectural concern is the placement of the system management functions and whether transportability and reusability could be enhanced by using generic representations and placing the majority of these functions within the respective layers as opposed to the present centralized

6.2.2 Ada Language Education

For the Ada language to achieve many of its goals, comprehensive training and education programs are needed at both management and technical levels. At the management level, it will be necessary for managers to become familiar with Ada concepts and changes in methodology. These in turn will influence design costs and the proportions of time devoted to the requirements, design, code and integration phases because of transportability and reusability considerations. At the technical level, attempting to teach Ada by giving a one-week syntax-oriented course, without the proper basis in software engineering will be of little help and may be counter productive due to the misuse of Ada features. A comprehensive course which addresses software engineering objectives, the Ada language features and how they can be used to support the objectives, and the role of using a well-defined design methodology and compatible, supporting tools will be prerequisites for producing personnel who are truly proficient. An important point will be the understanding of the PDL to be used and what is to be achieved through its use. An integral part of this training should include the use of the Rationale for the Design of the Ada Programming Language /HONEY84/ since it provides an important context for the purpose and use of the Ada features. Since the start of the project, considerable literature on the Ada language has been produced and should be selectively included in any training program. To the extent that transportability and reusability are major considerations, additional training and education will be required since these software characteristics will not be automatically achieved merely by using Ada.

6.3 SOFTWARE ARCHITECTURES

This section provides recommendations on alternatives to be considered and the use of generic architectures for the communications protocols and trusted software applications.

combinations of features were not used. Significant difficulty was encountered in the use of exceptions and in the use of global data with tasks. Due to the inherent properties of exceptions, two specific recommendations are made. It is recommended that total system error management, in terms of using Ada exceptions, be an integral part of the design process from the beginning of the macroscopic design phase. Since exceptions which occur in a package specification are only weakly associated with their source, it is recommended that supplementary ANNA information be provided to achieve a strong association of an exception with its sources.

Although much has been achieved through the standardization of Ada, there are still many elements associated with the use of Ada which have not been standardized.

One such characteristic of Ada not encountered with other languages is the close coupling between Ada features and the run-time support environment. Functions which were previously performed explicitly, such as task context switching, are now performed implicitly by the underlying run-time support environment. This is further complicated by the fact that in these instances the particular mechanization is NOT specified as part of the Ada standard with the result that any of several mechanizations may exist, some of which may be acceptable, some of which may not be. Other features such as the selective wait statement mechanization are implementation dependent. The issue of compiler pragmatics, such as stack size limitations, levels of nesting, size of packages and other factors lie outside the language and are not specified as part of the language. To reduce implementation problems, it is recommended that a set of evaluation criteria, based in part on application specific criteria, be used to select a compiler so that suitable information can be obtained on those areas left to the discretion of the compiler implementor.

because of the advanced type abstraction and software architecture definition features of Ada compared to other languages such as FORTRAN, C, and even PASCAL, and the ability to directly expand the Ada PDL specifications through the use of ANNA, that the previous distinction between requirements and designs, particularly formally stated requirements and designs, are no longer as distinct as they were. Consequently, until further experience is gained and definite structural and functional objectives are formed for the organization of the FTLS using Ada PDL and ANNA, the exact placement of the related activities will remain an open issue. Thus, for example, the DTLs and FTLs could be produced during the requirements phase; the design phase or the DTLs could be produced during the design phase; and the FTLs, and possibly one or two lower levels of detail, produced during the macro design phase.

6.2 ADA LANGUAGE

This section provides recommendations on the use of selected Ada-language features and on Ada education.

6.2.1 Ada Language Features

Because of the general difficulty in using the MIL-STD-1815A, /M18183/, an abridged version of the manual should be produced. This version should, at the very least, be more readable from a user standpoint, as opposed to that of a compiler implementor. It may be incomplete in the sense that all semantics of each syntactical form and all interactions between all language features need not be addressed, but rather referenced to the unabridged manual. This would enable individuals to learn Ada and become effective in its use much more rapidly and to still be aware of and able to locate necessary details and subtleties when required.

Because of compiler limitations, all Ada features were not used on the project. Such features as generics, task types, the use of allocators for creating tasks dynamically, and many other

effective by providing new information to be used for program analysis, eliminating manual efforts, reducing development time, eliminating various types of errors, or making the software interphase transitions smoother. The recommended tools are given in Table 6.1-1. A complete description which includes the phase to which the tool applies, purpose, functions, problems addressed, and the rationale for the tool are contained in Appendix C.

Table 6.1-1. Software Tool Recommendations

- o PDL Processor
- o Pretty Printer
- o Source-Level Debugger
- o Expanded Name Generator
- o Multi-Mode Syntax Directed Editor
- o Task Call Sequence Analyzer
- o Advanced SKETCHER
- o Ada-Preprocessor for Trusted Software Restrictions
- o Annotated Aad (ANNA) Compiler
- o Annotated Ada (ANNA) Run-Time Verifier

6.1.7 Trusted Software Development Methodology

Since the recommended methodology provides for stepwise refinement, it can naturally be applied to the development of trusted software. To accommodate the trusted software requirements, it is necessary to develop the Formal Top Level Specifications, Descriptive Top Level Specifications and their lower level extensions denoted as FnLS and DnLS. Previously the FTLS or its equivalent has been written in a language such as SPECIAL or GYPSY during the requirements phase. This specification was then extended via refinement until sufficient detail was present to permit coding in another language. As was the case on this project, translation and interpretation problems occur with this approach. To eliminate these problems, it is recommended that the Ada PDL of the trusted software be augmented with ANNA (Annotated Ada) to produce formally verifiable designs and implementations and that the ANNA be carried through the microscopic designs and into the code, if required, to produce a verifiable implementation. It should be noted that

6.1.4 Integrate/Test

In the classical software development approach, the integrate/test phase is the first time that all the software is brought together. This results in detecting module interface problems and adverse module interactions for the first time with the consequence that redesign and recoding of portions of the systems are required. With Ada and with the macro/micro design methodology, it is not necessary to wait until the integrate/test phase to bring the system components together. By compiling the entire set of compilation units and possibly including null package bodies to accommodate the placement of all context clauses, it is possible to achieve an initial degree of system integration at the conclusion of the macroscopic design phase. An executable PDL should be considered to achieve limited execution of the designs very early in the development effort.

6.1.5 PDL Considerations

It is recommended that reviews be conducted following the formation of the virtual packages, following the formation of the object-oriented design diagrams, and following completion and compilation of the macro and micro PDL. Such reviews serve the normal functions, as well as to avoid two basic problems. Reviews assure that coding is not performed in the macroscopic design phase where the emphasis should be on the system architecture; and that system architecture designs will not be left for completion during the coding phase. Careful attention should be given to the characteristics of the embedded English used in the PDL to assure that it is neither too detailed nor too abstract; otherwise, too many details will need to be supplied at another level of design.

6.1.6 Software Tools Recommendations

This section recommends generic software tools which should be reviewed for implementation. These tools would contribute toward making the software development methodology more efficient and

6.1.2 Microscopic Design Methodology

The microscopic design level follows naturally from the macroscopic level and readily permits refinements of the designs. Coding of generics is permitted since they will generally be required early in the code/debug phase.

An integral portion of the microscopic design is to include all known calls within the respective bodies whether they are to other entities within the package body or to entities supplied by another package of the same or other virtual package. This level of detail assures that library unit specifications are correct before the actual coding process is initiated. This has been found to be particularly important with respect to overall development efficiency. To the extent that library units need to be reorganized and recompiled during the code/debug phase, there may be significant ripple effects which impact on schedules and the availability of computing resources.

6.1.3 Code/Debug

The majority of the tasks in the communications protocols application were initially activated in the code/debug phase. The result was that both the tasking interface and control aspects as well as the computational aspects had to be dealt with at the same time.

In retrospect, it appears that a more effective approach would have been to achieve some type of executable integration of the tasking "shells" during the design process or very early in the code/debug phase. This could be achieved by an executable PDL processor or by simply forming task skeletons with the majority of the internal code suppressed. Using this approach, concurrent processing aspects involving control and data flow could have been isolated from the internal computational activities of the tasks and debugging and system integration could have been simplified.

SECTION 6

RECOMMENDATIONS

6.1 SOFTWARE DEVELOPMENT METHODOLOGY

This section provides recommendations on the macroscopic and microscopic design, code/debug and integrate/test phases and on PDL-usage considerations, software tools and a prototype trusted-software development methodology.

6.1.1 Macroscopic Design Methodology

The macroscopic design methodology works very well and should be formally documented and published. The combination of initial graphics followed by corresponding detailed text appears to be correct to achieve the top-level designs. The methodology avoids extraneous detail, yet provides a PDL as a suitable refinement and extension once the major design components have been established. The existing design guidelines focused primarily on assuring that the correct type of Ada-based information was provided. The guidelines should be expanded to address specific transportability and reusability considerations since such requirements need to be addressed from the outset of the design.

In the classical software development lifecycle, the development phases are disjoint from each other in that the design phase is initiated only after the requirements phase has been completed. This approach permits little opportunity for feedback based on insights gained in forming designs for the given requirements and results in discontinuity across the various phases. To minimize this problem, it is recommended that each succeeding phase overlap with its successor. This permits rapid prototyping at the design level to obtain additional information in areas where potential problems exist or the designs are very complex.

methodology and tools, robust compile-time environments will be required.

A second factor is compiler characteristics. It is essential to carefully review and understand Appendix F, Implementation-Dependent Characteristics, before beginning any design or programming effort. It will be prudent to formulate or obtain a compiler evaluation checklist regarding such features as compiler pragmatics (level of nesting of packages permitted, maximum length of identifiers, maximum number of tasks active, maximum number of task callers which can be queued) to assure that these features or limitations are well understood. Any design or programming restrictions can then be stated as part of some overall software development guidelines. Such lists are presently available in preliminary form.

5.6.2 Run-Time Environment

In the run-time environment, many of the same arguments apply as in the compile-time environment. Unfortunately, there is no specific place for documenting the information such as Appendix F. Again, a run-time environment checklist needs to be formed which reflects both general considerations as well as application-dependent characteristics. Such lists are presently available in preliminary form.

There were also several situations in which errors could occur but for which no exception handlers had been declared. This occurred primarily in the communications protocols application where two factors seemed to be at work which were lack of understanding of how exceptions functioned and how they were to be used. There was also a lack of detail in identifying error conditions as part of an overall software design process which results in programs working correctly with correct data, but being weak in integrity, reliability and robustness when erroneous data is encountered. By contrast, exceptions were much more an integral portion of the trusted software design since they were identified as part of the SPECIAL specifications.

In conclusion, the treatment of error conditions as integral to the design effort must be required, and these error conditions must be related directly to the use of Ada exceptions. To the extent that very complex architectures are formed, many Ada features are combined to achieve a design, and error management has not been an integral part of the design process, there will be an increased likelihood of encountering errors at the architectural level in the form of reduced robustness, integrity, and reliability. In addition, the likelihood of having erroneous programs may also increase.

5.6 PROGRAMMING SUPPORT ENVIRONMENT

This section presents conclusions relating to the compile-time and run-time environments.

5.6.1 Compile-Time Environment

The intent of Ada is not merely to present a new language for developing embedded computer systems; rather it is to provide a suitable tool which will permit broader software engineering goals to be achieved. Since such goals are dependent on software

architectural experimentation and tuning should be performed to determine the magnitude of changes in performance resulting from such changes. Fourth, based on the system testing and tuning results, it may then be possible to form refined design and programming guidelines which deal more specifically with the software performance quality factors.

5.5 SOFTWARE ERRORS

Software errors can be divided into three categories: those related to inexperience with the Ada syntax and semantics; those occurring at a somewhat higher level related to particular software architecture characteristics which result in erroneous Ada programs; and those occurring as a failure to use exceptions.

The number and types of errors diminished with time as experience with the limitations of the compiler became better understood. There was also the tendency to correct errors without fully understanding their cause, only to have the same type of error occur later at another place. Once this approach was detected, specific actions were taken to assure that not only was the error corrected, but also that the basic nature of the error and the associated language syntax and semantics were more completely understood as a way of avoiding the same error later. Constant referral to the reference manual may be required when initially learning the language in order to adequately understand the nature of the error and the appropriate corrective action.

There were some errors detected which manifested themselves in the form of erroneous programs. These errors were detected during design reviews and as a result of attempting to resolve other execution-related problems. In all instances, these errors were due to improper references to global data using tasks or hidden tasks and visible subprograms.

In the trusted software, an attempt was made to have the classification code of the data affixed to it in some way so that it could not be improperly modified. The types private and limited private seemed ideal to this purpose. However, the problem arises that a record declaration that contains a private or a limited private object, becomes itself a private or a limited private type. To use a record type that is private or limited private would require another level of complexity. For this reason, the GUARD does not contain any of these types.

The use of variant records proved valuable as a feature of Ada that would allow a single record type declaration the ability to define several different message formats in one structure. The resulting objects could have different characteristics for different uses, all under software control. In the case of the GUARD, the various transaction types, as defined in the requirements documentation, could use common interfaces and yet have totally different characteristics. The problem arises when the readability factor is considered. Embedded case statements are needed to define objects that can be common to some formats and not to others. The scoping rules for these 'case' statements are different from 'case' statements in the program bodies and can thus present maintainability problems.

5.4 SOFTWARE PERFORMANCE

Unfortunately, only a small portion of what was planned in the performance evaluation area was achieved due to the compiler problems. To conduct any meaningful performance tests several prerequisites need to be achieved. First, the originally intended designs need to be implemented, debugged, and integrated using a validated, full-capability Ada compiler. Second, moderately extensive system testing should then be performed to assure that the desired Efficiency-II, Integrity, Reliability, and Robustness software quality factors have been achieved. Third, following the system testing and, based in part on the system testing results,

what was originally intended. Thus, two prerequisites for conducting an evaluation of the trusted software are to implement the original designs and conduct extensive static and dynamic testing.

5.3.2.4 Compilation Unit Statement Characteristics

Some generalities concerning the trusted software are as follows:

COMPILATION UNIT SIZE	AVERAGE	LARGEST	SMALLEST
Code Statements	141	560	12
Comment Statements	199	1566	17
Total Lines	444	2555	38

Trusted Processes	25%
Non-trusted Processes	17%
Inter-Process Communication	9%
ARPANET Connection Emulation	17%
Support and Utility Modules	32%

With an arbitrary guideline of 1,000 lines per package, 4 out of 26 exceeded this guideline. If the separate compilation feature of Ada had been implemented, these four compilation units would have been smaller. There are 30 tasks operating in the GUARD; this is one each for the IPCs, the file handlers, and the locks. These tasks were implemented as data monitors and transaction transport tasks. There is one task in each of the independent trusted and non-trusted processes. Each functional process operates independently of the others. The ARPANET emulators contain two tasks, one to drive the MMI and the other to communicate asynchronously with the rest of the GUARD.

5.3.2.5 Other Observations

The workarounds needed to make the GUARD execute added considerable complexity to the system. The SPECIAL PDL was too complex, on the one hand, and too vague on the other. The Downgrade Trusted Process package could have been better organized if the specifications were not followed as strictly as they were.

impacts of exception management assessed. Due to the tight coupling between Ada and the run-time environment, and the many implementation-dependent options, it is necessary to understand how these features are mechanized and what their limitations are. Failure to do so may cause considerable problems with features such as recursive subprogram formulations, dynamic instantiation of generics, use of allocators, and other features which interact directly with the run-time environment. To the extent that these aspects are not known or not understood during the design process, it is recommended that selective prototyping be performed to gain the necessary information.

6.5 SOFTWARE ERRORS

Errors fall into two categories, which are the normal programming errors and those relating to overall software architectures. Many of the normal programming errors seem to be directly related to the nature of the Ada syntax and semantics and thus can be overcome via proper education, training, and experience. This will be particularly important when many Ada features are combined into a single design and interactions between features occur.

The software architectures themselves may have an influence on the types and numbers of errors which occur. Under such circumstances, architectural guidelines may have to be established to diminish the number, and severity of errors or to assure that errors are handled properly. One aspect that needs emphasis is that exception management must be an integral part of the design process. To the extent that software architectures are complex, it may become necessary to develop specific software tools to check, both statically and dynamically, for various types of error conditions. The architectures developed here did not appear to be sufficiently complex and insufficient performance testing was conducted to determine if such tools could have helped.

6.6 PROGRAMMING SUPPORT ENVIRONMENT

This section provides recommendations with respect to both compile-time and run-time factors which may impinge on a software development project as well as the overall quality and capability of the Ada programming support environment.

6.6.1 Compile-Time Environment

Because of the latitude afforded Ada compiler implementors, it is recommended that a set of compiler evaluation criteria be formed which is, in part, application dependent. These criteria should address the following three categories. First, the implementation-dependent features, such as the granularity of values for task priority should be evaluated. Second, the compiler pragmatics which may influence the length of identifier names, level of nesting and other factors which may impact a particular application or set of software development standards should be evaluated. Third, features outside the language specification, such as how task context switching, TEXT_IO, selective waits, and other features are mechanized, and the maximum number and size of tasks and maximum size of compilation units should be evaluated. Although such evaluation criteria may not solve all compile-time related problems, a careful assessment should significantly reduce the number and severity of problems. This evaluation should be made prior to performing any design so that constraints can be identified and understood.

6.6.2 Run-Time Environment

Because of the nature of Ada, many features which were previously directly provided by the run-time support system of the executive or operating system, are now generally hidden from direct view by Ada features such as allocators, unchecked deallocation and tasking features. To avoid a negative impact of particular mechanizations of these features, a set of run-time support environment criteria

should be formed and used in selecting a total Ada run-time environment. This evaluation should be made prior to performing any design so that constraints can be incorporated or addressed at the beginning of the design phase.

6.6.3 Programming Support Environment

The Ada programming support environment used on this project was a minimum environment in that the only available tools consisted of the prototype, partial implementation Ada compiler; SKETCHER; and EDT, the screen-oriented text editor. As full-capability compilers are developed and used, and as methodologies are formalized and supplemented with tools to make them effective and efficient, the programming support environment data storage and processing requirements will increase substantially. To realize Ada's promise of producing software which is more cost effective, it is recommended that significant effort be devoted to determining the processing and storage requirements at the outset of the project and that these assessments directly address the use of planned software development and maintenance tools.

6.7 PROJECT RECOMMENDATIONS

Although a great deal was accomplished in both application areas, several significant items were not accomplished because of compiler problems. The majority of these items falls under the category of performance testing and assessing the Correctness, Efficiency-II, Integrity, Reliability and Robustness. For both applications, a compiler should be selected which has been validated and evaluated to the specific features required for this type of application so that many of the previous problems can be either eliminated or reduced in severity and impact. The existing programs should be restored to their original designs and the implementation should be completed.

For the communications protocols, the previously planned software performance testing should be conducted after the originally planned implementation has been completed. The overall architecture should be evaluated with respect to the performance aspects and with regard to design and programming criteria for achieving transportable and reusable software.

For the trusted software, the previously planned software performance testing should be conducted after the originally planned designs have been implemented. This performance testing should include extensive stress testing, code analysis, and correspondence testing to the degree that such testing can be performed given the nature of the SPECIAL requirements for the Upgrade and Downgrade Trusted Processes and the emulation of KSOS.

An alternative approach would include selecting a compiler as indicated above but with a different development and evaluation approach, which would consist of the following steps. Obtain the revised set of trusted software requirements for the ACCAT GUARD which were formed in GYPSY /KEET81A/, /KEET81B/, /KEET81C/. Determine a suitable place in the requirements hierarchy to begin and apply the previously recommended trusted software methodology and guidelines, and use Ada and ANNA from the outset of the development and redesign and reimplement the trusted software while, in general, reusing the existing nontrusted software. Evaluate the software with respect to Maintainability, Testability, Correctness, Efficiency-II, Integrity, Reliability, Robustness and formal verifiability by performing extensive static and dynamic testing. Evaluate the design and programming guidelines used and revise them accordingly. Assess the impact of restrictions on the Ada language with regard to usability of the restricted subset.

SECTION 7
REFERENCES

The following references apply totally or partially as cited throughout this document.

7.1 MILITARY STANDARDS AND SPECIFICATIONS

/M16778/

Department of Navy, Military Standard, Weapon System Software Development, MIL-STD-1679 (Navy), 1 December 1978.

/M84773/

Military Standard - Format Requirements for Scientific and Technical Reports Prepared By or For the Department of Defense; MIL-STD-847A, 31 January 1973, including update notices 1 and 2.

/M18183/

United States Department of Defense, "Reference Manual for the Ada Programming Language," ANSI/MIL-STD-1815A-1983, 17 February 1983.

/M17783A/

Department of Defense, Internet Protocol, MIL-STD-1777, 12 August 1983.

/M17783B/

Department of Defense, Transmission Control Protocol, MIL-STD-1778, 12 August 1983.

/M15272/

Military Standard - Technical Reviews and Audits for Systems, Equipments and Computer Programs; MIL-STD-1521A (USAF), 1 June 1976.

/M49068/

Military Standard - Specification Practices; MIL-STD-490,
30 October 1968.

/48379/

Military Standard - Configuration Management Practices for Systems,
Equipment, Munitions and Computer Programs; MIL-STD-483 (USAF),
21 March 1979.

/DSDS83/

Joint Policy Coordinating Group on Computer Resource Management,
Computer Software Management Subgroup, Defense System Software
Development (DOD-STD-SDS), Proposed Military Standard, 5 December
1983.

7.2 SYSTEM SPECIFICATIONS AND REFERENCES

/WOOD78/

J.P.L. Woodward, "ACCAT GUARD System Specification (Type A)," MTR-
3634, The MITRE Corporation, Bedford, MA, August 1978.

/LOGI79A/

Logicon, "Formal Specification of GUARD Trusted Software (Draft),"
ARPA-78C032303, September 1979.

/LOGI79B/

Logicon, "ACCAT GUARD Program Development Specification (Type B5),"
ARPA-78C0323-01, February 1979.

/BALD79/

David L. Baldauf, "ACCAT GUARD Overview," The MITRE Corporation
(MTR-3861), Bedford, MA, November 1979.

/WEST79/

Western Union, "Initial AUTODIN II Segment Interface Protocol (SIP) Specification," (System Engineering Technical Note TN 78-07-31), DCA 200-C-637-P003, 5 March 1979.

/WEST78/

Western Union, "AUTODIN II Design Executive Summary," Western Union Telegraph Company, McLean, VA 22101, 18 May 1978.

7.3 OTHER GOVERNMENT REFERENCES

/USDO80B/

United States Department of Defense, "Requirements for Ada Programming Support Environments," "Stoneman," United States Government, February 1980.

/USDO83/

Department of Defense Trusted Computer System Evaluation Criteria, 15 August 1983, CSC-STD-001-83, Library No. S225,711.

7.4 NONGOVERNMENT REFERENCES

/BBNI76/

Bolt, Bernek, and Newman, Inc., "Development of a Communications Oriented Language, Parts I and II," Report No. 3261, 20 March 1976.

/SRII78/

SRI International, "Verification of Communications-Oriented Language Programs," SRI International Final Report, Project 6413, August 1978.

/HALS77/

Maurice H. Halstead, Elements of Software Science, Elsevier North Holland, Inc., New York, 1977.

/COOP79/

John D. Cooper and Matthew J. Fisher, Editors; Software Quality Management, Petrocelli Books, Inc., New York, 1979; "An Introduction to Software Quality Metrics," by James A. McCall.

/KEET81A/

Jim Keeton-Williams, Stanley R. Ames Jr., Bret A. Hartman, Ronald C. Tyler, "Verification of the ACCAT-GUARD Downgrade Trusted Process, Volume 1: Overview and Major Results," (MTR 8463), Volume 1, September 1981, The MITRE Corporation.

/KEET81B/

Jim Keeton-Williams, Charles H. Applebaum, "Verification of the ACCAT-GUARD Downgrade Trusted Process, Volume 2: Verification Theory," (MTR 8463), Volume 2, September 1981, The MITRE Corporation.

/KEET81C/

Jim Keeton-Williams, Bret A. Hartman, James Abbas, Ronald C. Tyler; "Verification of the ACCAT GUARD Downgrade Trusted Process, Volume 3: Specification and Proof," (MTR 8463), Volume 3, January 1982, the MITRE Corporation.

/KRIE83/

Bernard Krief-Bruckner, David C. Luckham, Friedrich W. von Henke, Olaf Owe, "Reference Manual for ANNA, A Language for Annotating Ada Programs (Preliminary Draft)," March 1983.

/LUCK84/

David C. Luckham, "On the Design of ANNA: A Specification Language for Ada," Computer Systems Laboratory, Stanford University, Stanford, CA 94305

/CHEH80/

M.H. Cheheyl, M. Glasser, G.A. Huff, J.K. Millen, "Secure System Specification and Verification: Survey of Methodologies," 20 February 1980.

/BRIN81/

Alton L. Brintzenhoff, Steven W. Christensen, David T. Moore, J. Marc Stonebraker, "Evaluation of Ada as a Communications Programming Language," Report DCA100-80-C-0037, 31 March 1981, NTIS AD-A-121938.

/BUHR84/

R.J.A. Buhr, System Design with Ada, Prentice Hall, Inc., Englewood Cliffs, NJ 07632, 1984.

/BOOC83/

Grady Booch, Software Engineering with Ada, The Benjamin/Cummings Publishing Company, Inc., California, 1983.

/PRIV82/

J.P. Privitera, "Ada Design Language for the Structured Design Methodology," in Proceedings of the Ada TEC Conference on Ada, 6-8 October, 1982.

/HONE84/

Honeywell, Systems and Research Center, 2600 Ridgway Parkway, Minneapolis, MN 55413; Alsys, 29 Avenue de Versailles, 78170 La Celle Saint Cloud, France, Rationale for the Design of the Ada Programming Language (Draft for Editorial Review), January 1984.

APPENDIX A
SOFTWARE DEVELOPMENT GUIDELINES

The following material is excerpted from the Draft Software Development/Management Plan, CDRL 002, 4 May 1983. The prefixed notes indicate the evaluation of the material for the Communications Protocols (CP) and the Trusted Software (TS) applications, respectively, according to the following criteria: 1-used, effective; 2-used, ineffective; 3-not needed, not used; 4-needed, not used; 5-new addition; 6-change to existing guideline.

3.6 SOFTWARE DEVELOPMENT GUIDELINES

3.6.2 Software Architecture Design Guidelines

The objective of these design guidelines is to assure that reasonable software engineering principles are used in forming the designs from the very highest level of abstraction downward. An additional and equally important objective is to assure that Ada features which can support and influence high-level software architectures are highly visible and understood. Specific software engineering principles which will be used in an Ada context are abstraction, information hiding, modularity, localization, uniformity, completeness and confirmability as presented in /BOOC83/.

From virtually the outset of the design, the use of certain Ada features such as package, subprogram and task specifications will be emphasized to assure that Ada capabilities which support the software engineering principles are well understood and are maximally used to influence the design in a favorable way. One design abstraction which will be used is the concept of a virtual package, a level of abstraction one level higher than an actual package, but drawing on the same concepts embedded in actual packages. Thus, at a high level of design, the architecture will already have an orientation toward Ada which hopefully will simplify the stepwise refinement process which will produce the macroscopic and microscopic designs and, finally, the code.

In documenting the designs at the virtual package and package level, the object-oriented-design diagrams of /BOOC83/ and /BUHR84/ will be used as a basis for producing the precursors of the macroscopic designs which will exist in Ada textual form. If instances occur where these notations are inadequate, then they will be augmented with additional notations or variations on an ad hoc basis after due consideration of the surrounding circumstances.

3.6.3 Ada Program Design Language Guidelines

For uniformity and consistency, the level of detail specified for the macroscopic and microscopic designs in Section 3.2.4, Levels of Design, has been extracted and repeated here. Additional details have also been included.

3.6.3.1 Macroscopic Design Guidelines

The following levels of detail will be produced for the macroscopic designs:

TS	CP	
[1]	[1]	1) Provide identification of all library and secondary units.
[1]	[1]	2) Provide identification of all visible package components.
[1]	[1]	3) Provide identification of all formal parameters for task entries, subprograms, and generic declarations.

[TS & CP - Too Early In the Design Process. In all probability, formal parameters will change somewhere later in the design process.]

[1] [1] 4) Provide virtually complete specification of all visible types.

[TS & CP - Too Early In the Design Process.]

[1] 5) Provide identification of compilation unit dependencies.

[1] [6] 6) Specify major types and components within visible modules.

[CP - Change Specification => Identification]

[1] [6] 7) Specify major flow control logic within complex visible modules.

[CP - Change Specification => Identification]

[3] [1] 8) Declare all nested program modules.

[4] [4] 9) Identify all exception handlers.

[1] 10) Use English language text between brackets ([]) to indicate where conversion to code is required.

[1] [1] 11) Use comments, which can be retained in the code, to provide overviews and augment and clarify data structures and processing.

[1] [1] 12) Provide references to imported exceptions, subprogram and task calls and to local exceptions, subprograms and task calls at the next lower level of detail.

[5] [5] 13) Assign weights to software quality factors for each compilation unit.

[5] [5] 14) Compile all library units at the conclusion of the macroscopic design and remove any deficiencies.

3.6.3.2 Microscopic Design Guidelines

The following levels of detail will be produced for the microscopic designs:

TS	CP		
[1]	[1]	1)	Complete specification of all components of the visible and private portions of all library units.
[1]		2)	Complete specification of default initialization for all visible and private objects.
[1]	[1]	3)	Complete specification of all major flow control logic within all modules.
[1]	[1]	4)	Complete identification of local (inner scope) types and objects.
[1]	[6]	5)	Identify all nonvisible subprogram tasks.
[1]	[1]	6)	Refine major flow control within visible and complex modules.
[1]	[1]	7)	Use English language text between brackets ([]) to indicate where conversion to code is required.
[5]	[5]	8)	Compile all library units at the conclusion of the macroscopic design and remove any deficiencies.

3.6.4 Ada Programming Guidelines

The following sections specify nominal program design guidelines for the macroscopic and microscopic designs and for the programs themselves. Obviously, the Programming Guidelines must be considered not only at the programming level but also at the macro and micro design levels and, in some cases, even at the system architectural level.

AD-A152 314

EVALUATION OF ADA (TRADEMARK) AS A COMMUNICATIONS
PROGRAMMING LANGUAGE VOLUME 1(U) SYSCON CORP SAN DIEGO
CA A L BRINTZENHOFF ET AL. 01 MAR 85 DCA100-83-C-0029

3/3

UNCLASSIFIED

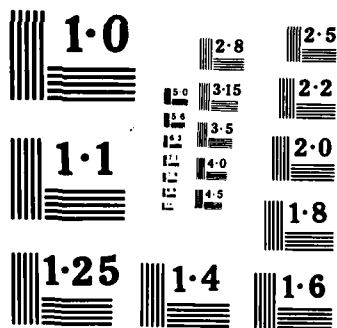
F/G 9/2

NL

END

FORMED

DATE



Because many of these guidelines are closely associated with the Ada syntax and other program module considerations, they have been placed at the programming level and oriented along the chapter titles of the Ada Language Reference Manual.

As the title indicates, these are guidelines and not standards or mandatory conventions. It is anticipated that because of individual programming styles, past experience, and special situations, variations and even some distinctly different conventions will be formed and used. The objective here is to avoid what appear to be bad choices, and emphasize nominal good choices so as to minimize overall program development problems. Clearly, one aspect of the software evaluation will deal with the formulation and use of standards, guidelines and conventions based on the experience gained in the use of the following guidelines.

3.6.4.1 Overall

TS	CP	
----	----	--

- | | | |
|-----|-----|--|
| [1] | [4] | 1) Use comments liberally: on statement lines to clarify code; as "headers" to introduce complex sequences of code; for the module summary which occurs inside the module including packages, subprograms, blocks, tasks, and entries, if appropriate. |
| [1] | [1] | 2) Use only a single statement per line except for comments and combined with and use clauses relating to the same compilation unit. |
| [1] | [1] | 3) Limit subprogram and task body sizes to approximately 100 lines of code, and a maximum of 200 lines. |
| [1] | [6] | 4) Limit text to 80-character lines. |

[CP - A rigid 80 column requirement must be traded off with overall readability due to the long names and expanded name notation referencing external units]

- [1] [6] 5) Use blank lines to provide separation of different entities such as one group of types/objects and another.

[CP - Or user defined separator lines]

- [1] [1] 6) Use recommended paragraphing of Ada Language Reference Manual.
- [6] [1] 7) Indent paragraphs in increments of three characters.

[TS - Used increments of 2 characters for indentation.]

- [1] [1] 8) Align subprogram, task entry, and generic formal parameters for easy reading.
- [4] [1] 9) Use page breaks (pragma PAGE) to begin major program units and to otherwise separate distinct, different entities.
- [1] [1] 10) Use lower case letters for reserved Ada words.

3.6.4.2 Lexical Elements

TS CP

- [6] [6] 1) Lexical order: (types(s), objects), packages, subprograms, tasks, exceptions.

[CP & TS - Ordering may be influenced by nesting, call dependencies, etc. Cannot always be adhered to as stated. Important aspect is to be consistent.]

- [1] [1] 2) Use the package, subprogram and task identifier in the respective end statements.
- [4] [1] 3) Place pragma statements so they are highly visible.

3.6.4.3 Declarations and Types

TS CP

[1,2] [1] 1) Use default initialization when initial values are required.

[1,2] [6] 2) Choose type and object identifiers for compatibility and meaningfulness.

[CP & TS - Add "whenever possible". Identifiers may be chosen to reference or be determined by external specification requirements]

[1] [1] 3) Use enumeration types for improved readability.

[4] [4] 4) Use derived types and subtypes for clarity and to avoid computational ambiguity.

[3] [1] 5) Keep recursive and mutually dependent type declarations in close proximity to each other.

[2] [5] 6) Avoid/minimize large and complex data types and objects.

[CP & TS - Data Structures may be dictated by external specifications and PDL]

3.6.4.4 Names and Expressions

TS CP

[1] [1] 1) Use a space before and after delimiters and compound delimiters except for: the apostrophe in attribute notation; the period in selected/expanded component notation; the comma separating array indexes and actual parameters; parentheses in array component selection.

[1] [1] 2) Avoid use of unnamed literals in executable code.

[1] [1] 3) Use attribute notation wherever appropriate to minimize explicit dependencies.

- | | | | |
|-----|-----|----|--|
| [4] | [1] | 4) | Use named associations for record and array aggregates unless the specified values are meaningful. |
|-----|-----|----|--|

3.6.4.5 Statements

- | TS | CP | | |
|-----|-----|----|---|
| [1] | [4] | 1) | Use blocks to provide localized exception handlers. |
| [2] | [1] | 2) | Use case statements instead of if statements when possible. |
| [2] | [4] | 3) | Use blocks to "collect" highly localized operations and corresponding data. |
| [3] | [3] | 4) | Provide meaningful block names for blocks. |
| [3] | [3] | 5) | Provide meaningful loop names for loops. |

3.6.4.6 Subprograms

- | TS | CP | | |
|-----|-----|----|--|
| [6] | [3] | 1) | Avoid recursive invocations. |
| [3] | [1] | 2) | Declare new operators only for frequent and meaningful use. |
| [1] | [1] | 3) | Use overloading of subprogram identifiers only when operations are highly similar or number of arguments vary. |
| [1] | [1] | 4) | Limit formal arguments to approximately five. |
| [1] | [1] | 5) | Choose formal parameter names which will be communicative if the named notation is used. |
| [1] | [4] | 6) | Itemize subprogram identifier and formal parameters, one item per line, followed by a descriptive comment. |
| [1] | [1] | 7) | Avoid use of functions which produce side effects unless absolutely required. |
| [1] | [1] | 8) | Subprogram identifiers and block and loop names should be meaningful where they are used, not where they are declared. |

3.6.4.7 Packages

TS CP

- [1] [6] 1) Minimize package nesting unless exceptional requirements exist.

[CP - Package Nesting should reflect architectural Requirements/Specifications]

- [1] [1] 2) Export only those package components which are absolutely required.

- [1] [6] 3) Use package identifiers which communicate the package purpose.

[CP - Names => Functional and/or Architectural Significance]

- [1] [1] 4) Within package bodies, declare data types/objects, nonvisible packages, subprograms, tasks followed by the visible subprograms and tasks.

- [1] [1] 5) Preserve lexical ordering of entities such as sub-programs, and tasks and entries between package specification and package body.

- [3] [1] 6) Use private and limited private types judiciously since they will require extra subprograms to provide the necessary user operations.

- [1,2] [1] 7) Visible entities of a package should have meaningful identifiers not merely generic names such as, for example, READ and WRITE.

3.6.4.8 Scope and Visibility

TS CP

- [1] [3] 1) Use use clauses for Ada-defined and locally declared units only.

- [1] [1] 2) Use expanded name notation for components of external, user-defined compilation units.

- [1] [1] 3) Use renames to simplify notation or to abbreviate expanded names such as package identifiers.
- [1] [1] 4) Avoid complex scope, visibility and overloading relationships which produce hiding or other disjoint visibility relationships.

3.6.4.9 Tasks

TS CP

- [1] [1] 1) Itemize task entry points and corresponding formal parameters, one item per line, followed by a descriptive comment.
- [1] [3] 2) Avoid use of abort.
- [1] [1] 3) Use entry names which are meaningful where they are used, not where they are defined.
- [1] [1] 4) Choose formal parameter names which will be communicative if the named notation is used.

3.6.4.10 Program Structure and Compilation

TS CP

- [6] [6] 1) Minimize complex or lengthy compilation unit dependencies.

[CP & TS - The choice/complexity is not always user determined]

- [4] [4] 2) Use subunits and secondary units to control the size of compilation units.

[CP & TS - Needed, Not Available]

- [6] [4] 3) Use judicious nesting of modules in order to simplify or minimize overall compilation dependencies and number of compilation units.

- [5] [5] 4) Place context clauses with package bodies whenever possible instead of with package specifications.

[CP - Needed, Not Available]

[TS - Original design required nested modules for file, port, and other utility packages in common packages for both the high and low sides of the GUARD]

3.6.4.11 Exceptions

TS CP

- [1,2] [6] 1) Use exceptions for truly abnormal error or exceptional circumstances and not to effect normal changes in control flow.

[CP & TS - Useful Debugging Tool]

- [6] [4] 2) Avoid the use of others in exception handlers since it can produce misleading results.

[TS - Useful Debugging Tool]

- [1] [1] 3) User-declared exceptions should not overload Ada-defined exceptions.

- [1] [1] 4) Localize exception handling as much as possible.

- [1] [1] 5) Use exceptions in tasks and especially in a rendezvous very cautiously.

- [1] [1] 6) Declare, in the package declaration, all exceptions of modules of packages which will be visible externally.

3.6.4.12 Generic Units

TS CP

- [2] [3] 1) Limit generic parameters to approximately five.
- [1] [3] 2) Choose formal parameter names which will be communicative if named notation is used.
- [1] [3] 3) Avoid nesting of generics.
- [2] [3] 4) Use extreme caution and explicit documentation if generic parameters are used as conditional compilation flags.

3.6.4.13 Representation Clauses

TS CP

- [3] [1] 1) Place representation clauses in close proximity to the components to which they apply.
- [3] [1] 2) Use unchecked operations cautiously and mark their locations to be highly visible.

3.6.4.14 Input-Output

TS CP

- [6] [6] 1) Centralize column, line and page specifications for text files if possible.

[CP & TS - Centralize all I/O operations/routines in separate units (packages) according to the type of I/O (ie Disc, CRT/Printer etc.)]

APPENDIX B
ADA RESTRICTIONS FOR TRUSTED SOFTWARE IMPLEMENTATION

The following is a preliminary list of restrictions to be imposed on the Ada language for the purpose of developing trusted software. These restrictions were formed based on considerations of maintainability, testability, correctness, integrity, reliability, robustness, and formal verifiability of the software itself and on retaining a basic usability of the Ada language in the context of these restrictions.

These guidelines or restrictions form a preliminary list which is based, in part, on some limited Ada programming of trusted software on a prototype basis. However, this software is incomplete and has not been implemented along the lines of the actual designs because of limitations with the prototype compiler which was used. Thus, because extensive stress testing, covert channel analysis, and correspondence analysis of the code have not been performed, the guidelines have not been evaluated or verified with the result that they may be incorrect, incomplete, inconsistent or inappropriate at this point.

1. General Considerations

- 1) Prohibit use of all features or combinations of features which result in erroneous programs regardless of whether such programs are in fact incorrect or not in the given implementation.

2. Lexical Elements

3. Declarations and Types

- 1) Prohibit use of anonymous types.
- 2) Assign default values to all objects except limited private objects which should be preset via explicit initialization in the executable portion of a body.
- 3) Prohibit trusted-non-trusted process communication from using access variables.

4) Prohibit record types and records of the respective types from becoming private or limited private objects as a side effect of containing objects which are of a private or limited private type.

4. Names and Expressions

5. Statements

1) Prohibit the use of goto statements as the predominant means of effecting changes in control flow.

2) Prohibit use of conditional exit statements which depend on global objects.

6. Subprograms

1) Attempt to achieve pass-by-copy as opposed to pass-by-reference variable transfers especially between trusted and non-trusted entities.

2) Avoid the use of subprogram declarations and corresponding bodies or subprogram bodies as compilation units (i.e., place subprograms inside packages except for the main subprogram).

3) Permit only procedures to produce side effects and prohibit functions from producing side effects.

4) Prohibit aliasing of in-out mode parameters in procedures.

5) Prohibit functions from using external global parameters unless they are constants.

7. Packages

1) Prohibit declaration of visible objects in the package specification for packages visible to non-trusted software.

2) Prohibit non-trusted software from declaring objects in its space which are manipulated by the secure package since unchecked conversion can then be used.

3) Prohibit accessing of global parameters from within a package body.

8. Visibility Rules

1) Prohibit renaming task entries as procedures.

9. Tasks

1) Prohibit use of guard conditions on task entries where global parameters are components of the conditions; permit only local parameters to be used on guard conditions.

2) Prohibit use of shared variables unless the tasks and shared variables are declared in the same scope.

3) Permit task types only when they are used to create a finite number of declared task objects.

4) Prohibit task types from being visible to a non-trusted process since the user can then create his own tasks.

5) Permit a task to abort only itself.

6) Prohibit the use of tasks where tasks would be used to achieve elaboration of some data other than that contained within the task itself.

7) Permit the activation of tasks via allocators only in declarative regions.

8) Prohibit the reassignment of tasks among the set of access variables designating those tasks.

10. Program Structure and Compilation Issues

LRM Ada Feature	E	I	O	P
13.7 Package System	X		N	
13.7 Memory_Size (pragma)				
13.7 Storage_Unit (pragma)				
13.7 System_Name (pragma)				
13.8 Machine Code Insertions		X	N	
13.9 Interface to Other Languages	X		N	
13.10 Unchecked Programming				
13.10.1 Unchecked Storage Deallocation	X		L/H	
13.10.2 Unchecked Type Conversion				
14. Input-Output				
14.1 External Files				
14.2 Sequential and Direct Files	X		H	
Task stack overflow when accessed from task				
14.3 Text Input-Output	X	X	H	
Synchronous vs asynchronous mechanization				
Impedes MMI operations in multiuser system				
14.3.9 Enumeration I/O			X	H
14.4 Input-Output Exceptions				
14.5 Package I/O_Exceptions (spec)				
14.6 Low Level Input-Output			X	H
A Predefined Language Attributes				
B Predefined Language Pragmas				
Optimize (pragma)			X	N
Page (pragma)			X	L
Source_Info (pragma)				
C Predefined Language Environment				
F Implementation-Dependent Characteristics				
Run-until-block context switch algorithm				H
Synchronous vs asynchronous TEXT_IO mechanization				H
Unrealistically small, fixed task stack size				H

LRM Ada Feature		E	I	O	P
9.	Tasks				
9.1	Task Specifications		X		H
	1. Must appear at outer-most scope				
9.1	Task Bodies		X		H
	1. Must appear at outer-most scope				
9.2	Task Types and Objects			X	H
9.5	Entries			X	M/H
	Entry families				
9.5	Entry Calls				
9.5	Accept Statements				
9.6	Delay Statements			X	M
9.6	Package CALENDAR	E	E		H
	Undefined exceptions				
9.7.1	Selective Waits				
9.7.2	Conditional Entry Calls			X	M
9.7.3	Timed Entry Calls			X	M
9.8	Priorities (pragma)			X	M
9.9	Task and Entry Attributes			X	M
9.10	Abort Statements			X	
9.11	Shared Variables (pragma)			X	M
10.	Program Structure and Compilation Issues				
10.1.1	With Clauses				
10.2	Subunits (is separate)			X	M/H
10.5	Elaborate (pragma)			X	N
11.	Exceptions				
11.1	Exceptions Declarations				
11.2	Exception Handlers				
11.3	Raise Statement				
11.4	Exception Handling				
11.7	Suppressing Checks (pragma)			X	N
12.	Generic Units				
12.1	Generic Declarations			X	H
12.2	Generic Bodies			X	H
12.3	Generic Instantiation			X	H
13.	Representation Clauses				
13.1	Representation Clauses			X	H
13.1	Pack (pragma)				
13.2	Length Clauses			X	H
13.3	Enumeration Rep Clauses			X	H
13.4	Record Rep Clauses			X	H
13.5	Address Clauses				
13.6	Change of Representation			X	H

LRM Ada Feature		E	I	O	P
4.	Names and Expressions				
4.2	Literals				
4.3.1	Record Aggregates		X	L/M	
4.3.2	Array Aggregates		X	L/M	
4.4	Expressions				
4.6	Type Conversion		X	L	
4.7	Qualified Expressions				
4.8	Allocators		X	H	
4.8	Controlled (pragma)		X	H	
4.9	Static Expressions				
4.9	Static Subtypes				
4.10	Universal Expressions				
5.	Statements				
5.2	Assignment Statement				
5.2.1	Array Assignment Statement				
5.3	If Statement				
5.4	Case Statement			L	
	1. Cannot incorporate attributes				
	2. Subtype or type must be in scope				
5.5	Loop Statement				
5.6	Block Statement				
5.7	Exit Statement	X		L	
5.8	Return Statement				
5.9	Goto Statement				
*6.	Subprograms				
6.1	Subprogram Declarations				
6.2	Formal Parameter Modes	X		L/M	
6.3	Subprogram Bodies				
6.3.2	Inline Expansion (pragma)		X	N	
6.4	Subprogram Calls				
6.4.2	Default Parameters		X	L	
6.5	Function Subprograms				
6.6	Overloading of Subprogram				
6.7	Overloading of Operators				
7.	Packages				
7.2	Package Specs and Decls.	X		N	
7.3	Package Bodies	X		N	
7.4.1	Private Types				
7.4.4	Limited Types				
8.	Visibility Rules				
8.4	Use Clauses				
8.5	Renaming Declarations				
8.6	Package Standard				

The following table indicates the possible types of deficiencies for each type of Ada feature (E-erroneous feature, I-incomplete/inconsistent feature, O-omitted feature, P-impact code) and the specific impact of that feature on the overall design and implementation (N-none, L-low(minor inconvenience, implementation (code) detail, M-medium (moderate inconvenience, some micro design changes, resort to less direct means), H-high (significant change in design at macro level or incurrence of significant additional work), T-terminal case (total impass regarding implementation of desired capability). In some instances supplementary notes are also provided. The compiler used was the prototype Telesoft-Ada* compiler, version V30R23 of November 1983.

LRM Ada Feature	E	I	O	P
*2. Lexical Elements				
*3. Declarations and Types				
3.2 Named Numbers (constants)				
3.3.1 Type Declarations		X		N
3.3.2 Subtype Declarations	X	X		N
3.4 Derived Types			X	L
3.5.1 Enumeration Types				
3.5.2 Character Types				
3.5.3 Boolean Types				
3.5.4 Integer Types				
3.5.5 Discrete Operations		X		L
3.5.7 Floating Point			X	N
3.5.8 Floating Point Operations		X		N
3.5.9 Fixed Point			X	N
3.5.10 Fixed Point Operations			X	
3.6 Array Types		X		H
Unconstrained array declarations				
3.6.2 Array Operations				
3.6.3 String Type		X		M
Unconstrained string declarations				
3.7 Record Types				
3.8 Access Types				
3.8.1 Incomplete Type Declaration				
3.9 Declarative Parts				

*Telesoft-Ada is a Trademark of Telesoft.

APPENDIX D
COMPILER LIMITATIONS AND IMPACTS

*Phase : Macro/Micro Design, Code/Debug

*Name : Annotated Ada (ANNA) Run-time Verifier

*Purpose: Provide run-time verification of specified ANNA constraints.

*Functions: 1) provide routines which perform run-time verification of constraints specified via ANNA

*Problems Addressed: Provide a means of verifying, during execution, that constraints which are specified via ANNA are actually being met.

*Rationale: Since extensive run-time testing will normally be conducted for trusted software, a software tool which automatically checks for violation of constraints specified in ANNA will improve both the quality of the software and the efficiency and effectiveness with which the software is tested.

*Phase : Macro/Micro Design, Code/Debug

*Name : Ada Preprocessor for Trusted Software Restrictions

*Purpose: Provide syntactical and semantical processing of Ada and/or Ada PDL to assure that the trusted software restrictions are enforced in the Ada PDL and Ada source code.

*Functions:

- 1) provide mechanization of trusted software restrictions
- 2) process Ada PDL and Ada source code to assure that syntactical and semantical trusted software restrictions are enforced.

*Problems addressed: Provide a software tool to effectively and efficiently assure that trusted software restrictions are strictly enforced.

*Rationale: Because of the need to impose restrictions on the Ada features which are used in implementing trusted software, it will be necessary to have a software tool which consistently, effectively and efficiently assures that the restrictions have not been violated.

*Phase : Macro/Micro Design, Code/Debug

*Name : Annotated Ada (ANNA) Compiler

*Purpose: Compile ANNA annotations embedded in Ada PDL and Ada source code.

*Functions:

- 1) Provide syntactical and semantical checking of ANNA source code and
- 2) provide code-generation capabilities for inclusion of ANNA Run-time Verifier software

*Problems Addressed: Provide a software tool to assure that the embedded ANNA is correct, complete and consistent.

*Rationale: The incorporation of ANNA into the Ada PDL and Ada of trusted software and other reusable or transportable software provides the capability to provide additional semantical information on the functionality of that software. This information can be relied upon only if it has been systematically error checked.

*Rationale: Considerable time and effort were devoted to refining the code that was produced which could have been more effectively devoted to an analysis of the designs and possibly the enhancement of the designs.

*Phase : Macro/Micro Design, Code/Debug

*Name : Task Call Sequence Analyzer

*Purpose: Provide the designer with textual or graphical representation of possible task call sequences or deficiencies.

*Functions: Provide a multi-level caller-callee tree which can be analyzed for potential starvation, deadlock, or unreachable callees.

*Problems addressed: A static analysis of complicated applications involving significant tasking could be performed in order to eliminate as many tasking errors or to at least identify possible problems prior to beginning execution.

*Rationale: Debugging of complicated tasking applications could be simplified and redesign could possibly be minimized by detecting "obvious" errors during the design phase.

*Phase : Macro Design

*Name : Advanced SKETCHER

*Purpose: Permit designers to produce top-level designs interactively using bit-mapped graphics and automatically convert the graphical representations into the corresponding skeletal PDL.

*Functions: Produce bit-mapped graphical representations of OODD's for inclusion in design documentation, provide generation of corresponding PDL.

*Problems addressed: Improved productivity during early design phase by elimination of manual activities.

*Rationale: Avoid dealing with excessive detail required in textual representations and provide automation for the straightforward process of converting the OODD's into skeletal PDL which can then be refined.

*Phase : Macro/Micro Design, Code/Debug

*Name : Expanded Name Generator

*Purpose: Provide the expanded names for entities referenced correctly by the compiler via use clauses.

*Functions: Prefix the correct expanded name to entities, such as the package name to a task contained in the package where the name is obtained by the compiler via a use clause.

*Problems addressed: Provide completely qualified entities through the inclusion of expanded names in order to facilitate code reading, traceability and maintainability.

*Rationale: During development it is convenient and efficient to not specify the expanded names, especially if they are lengthy; although renames clauses could be used for packages, for example, this may not be desirable. This would permit the development to progress quickly by eliminating the prefixing of lengthy names and would at the same time provide an effective way for including them after the fact to achieve the desired maintainability and traceability.

*Phase : Macro/Micro Design, Code/Debug

*Name : Multi-Mode Syntax Directed Editor

*Purpose: Facilitate the creation of the Ada PDL, the conversion of the PDL to code and the creation of Ada source code.

*Functions: Provide syntax directed editing, based on the particular mode selected (Macro, Micro, Code), facilitate conversion of English language entities into Ada or refined English language entities by supporting various display/prompting/relocation capabilities.

*Problems addressed: Assure that PDL or source code is produced which is more nearly complete, correct and consistent and provide for improved productivity.

*Phase : Code/Debug
 *Name : Pretty Printer
 *Purpose: Provide standardized formatting of source code
 *Functions: Provide standardized formatting of produced code regarding lexical format, placement of headers and comments.
 *Problems addressed: Eliminate the need for manual "pretty-print" formatting of code during original production as well as during revisions of the code during debugging and provide a method for standardizing the code produced by different programmers.
 *Rationale: Considerable time and effort can be saved through the use of a Pretty Printer for "structuring" the code into a more readable format and, at the same time, providing standardization in the source code produced by several individuals.

*Phase : Code/Debug, Integration/Test
 *Name : Source-Level Debugger
 *Purpose: Provide debug programs using source level information
 *Functions: Provide the ability for setting breakpoints and examining/setting values based on source code information as opposed to object code information. Provide features which are especially oriented toward tasking and enable the user to control which tasks are currently active or suspended and to ascertain the status of any activated, nonterminated task and the ability to examine various task queues to determine which tasks are runnable, suspended and possibly reorder task queues to resume a particular task or sequence of tasks.
 *Problems addressed: Provide a way for minimizing recompilation and the insertion of debugging code into the developed source code which must subsequently be revised.
 *Rationale: A source-level debugger can speed up the development cycle significantly as opposed to entering and deleting debugging statements which may have other adverse effects, especially in a tasking environment.

The following software tools are recommended as a minimum set based on specific experience during the project. It is believed that if even a subset of these tools had been available, a significant amount of time could have been saved. The summary of each tool indicates the phase in which the tool would be used, the (generic) name, purpose, functions, problems addressed, and rationale for the tool.

*Phase :	<u>Macro/Micro Design</u>
*Name :	PDL Processor
*Purpose :	Process and verify Ada PDL and provide various cross-reference information.
*Functions:	<ol style="list-style-type: none">1) provide well defined syntax/semantics for Ada PDL2) specify minimum acceptable requirements as per design phase3) verify completeness, correctness, consistency of PDL supplied4) provide various types of cross-references such as caller-callee relationships, type/object cross-references, and compilation unit caller-callee relationships
*Problems addressed:	Provide a software tool for assuring that the minimum required information is required and provide a way of presenting the information in alternative formats which will make various types of errors or deficiencies evident.
*Rationale:	A PDL processor will permit designs to be produced in an iterative fashion with refinement as details become known and are appropriate to the respective level of design abstraction. This will permit inappropriate details to be avoided and, at the same time, permit designs to be more complete and to be evaluated more thoroughly.

APPENDIX C
SOFTWARE TOOL RECOMMENDATIONS, DESCRIPTIONS

4) Prohibit instantiation of generics in which generic parameters are dynamically determined such as those passed into a block or subprogram.

13. Representation Clauses and Implementation Dependent Features

1) Encapsulate unchecked storage deallocation and assure overwriting in order to eliminate residual data before deallocation.

14. Input-Output

A Predefined Language Attributes

1) Prohibit use of attributes P'COUNT, P'CALLABLE, P'TERMINATED.

B Predefined Language Pragmas

1) Prohibit suppression of run-time checks within trusted software.

C Predefined Language Environment

F Implementation-Dependent Characteristics

1) Minimize use of implementation-dependent features since these features may change with compiler revalidation and thus require redesign, reimplementation and retesting.

11. Exceptions

1) Return input or default values on out, and in out mode parameters if an exception occurs and make no changes to these parameters until all results have been calculated correctly.

2) Indicate via ANNA or other annotation which exceptions are associated with which task entry or subprogram call. (Note that associating exceptions with entry points may not be adequate in the case where the same entry appears in multiple accept statements and different accepts produce different exceptions; this could be viewed as a design deficiency or could be addressed by providing annotated instances of multiple occurrences of the same entry point but with different exceptions.)

3) Prohibit use of predefined exceptions in place of user defined exceptions.

4) Provide a specific unique exception handler respectively for each unique exception which can occur.

5) Minimize propagation of exceptions by maximizing local processing of exceptions.

6) Use others clauses in exception handlers only to capture unexpected exceptions so that they can be dealt with explicitly and explicitly enumerate predefined Ada exceptions.

7) Assure that a package body, and its respective task and subprogram bodies, either remain valid after an exception has been propagated outside the package or that future calls elicit a suitable exception.

12. Generic Units

1) Use no generic formal objects of mode in out.

2) Prohibit aliasing of all generic actual parameters at instantiation.

3) Prohibit declaration of generic parameters which are subprogram specifications.

END

FILMED

5-85

DTIC